Code to Models

A Microcontroller from the Bottom Up

Andrew Mangogna

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	April 3, 2021	Start of document.	GAM
0.2	April 27, 2021	Part I first draft completed.	GAM
0.3	June 2, 2021	Further refinements to Part I. Part II draft completed.	GAM
0.4	July 15, 2021	Draft of background / foreground execution synchronization if place.	GAM
0.5	August 11, 2021	Crossing the Divide part is finished. Some document reorganization.	GAM
0.6	September 9, 2021	Added Batten Down the Hatches to configure the MPU.	GAM
0.7	September 18, 2021	Added Time Mancery to implement a timer queue which will be needed later.	GAM
0.8	October 17, 2021	Completed coding work on Time Mancery. Improved explanations in other parts.	GAM
0.9	October 30, 2021	Completed coding work on Tyranny of the Pins.	GAM
0.10	November 15, 2021	Reorganized book as three parts. Finished GPIO pin examples. Continuing refinement of text language.	GAM
0.11	November 30, 2021	Added chapter for UART device.	GAM
0.12	December 26, 2021	Completed work on UART device chapter. Writing for the first part of the book is complete. Additional copy editing is now required.	GAM
1.0	March 31, 2022	First release of the book which contains only Part I.	GAM

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Note
This book is a work in progress. This introductory chapter is incomplete. As of this version, Part I is substantially complete. Part II is a placeholder. It is a port of the `micca` run time to use the infrastructure described in Part I. Part II is intended to be rewritten to describe the run time execution model more in terms of the bottom up approach. Part III is empty. Part IV is a compendium of supporting software and included code is complete. However, more support code will be added.

This book contains a set of design concepts and their corresponding implementation for engineering software of reactive-oriented systems using semi-formal techniques[1]. Starting from well-founded principles, the book describes a path for implementing complex software systems that interact with their environment while abiding by sound software engineering principles. Along that path, the following broad areas are explored:

1. Basic systems programming required to run any software and have that software interact with its environment.
2. A model of software execution which both orchestrates control flow and enforces data consistency.
3. A composition technique that supports building software components and assembling those components into larger software systems.

In this book, there are several recurring software themes:

Separation of concerns
Understandability is a critical component of a sustainable software system. Our ability to conceive and understand complex systems is largely based on dividing the system into coherent subject parts. Those parts allow us to consider the aspects of the system in isolation. Implied in the separation is a technique to assemble the parts into a functioning system. This separation (and the subsequent reassembly) are necessary to overcome our human limitations of short term memory.

Constrained execution
There are many more things in a software system which can go wrong than can go right. The state space of even the smallest computing system is so overwhelming large that we must make efforts on several fronts to confine a program's execution and verify those constraints during execution in an attempt to minimize its failure modes.

Allocation of complexity
The idea to consider system complexity as consisting of *essential* complexity and *accidental* complexity was first proposed by Brooks. Semi-formal models of the logic of the system are the means by which the two forms of complexity are separated. Models capture system requirements and represent the system in an implementation independent form which is free of accidental complexity. Those models are translated into system software yielding any accidental complexity associated with the implementation. The solution to a problem must precede the implementation of that solution. The

[1] We use the term *semi-formal* to avoid any implication that we intend to prove program correctness in a mathematical sense. We leave mathematical proofs of correctness to Computer Scientists. We are satisfied here to be better Software Engineers.

fundamental difference between "what" a system does and "how" technology is brought to bear to implement the system is the basic division between the essential and the accidental. The view of complexity here is similar to that of Moseley and Marks.

These goals present a large scope of effort which must be narrowed substantially in order to accomplish them. We start by defining some fundamental terms.

We adopt the categorization of systems by Halbwachs:

a. Reactive — "systems that continuously react to their environment at a speed determined by this environment."
b. Interactive — "systems which continuously interact with their environment, but at their own rate."
c. Transformational — "classical systems whose inputs are available at the beginning of the execution and which deliver their outputs when terminating."

The focus of this book is *reactive* systems. Again, using Halbwachs definitions, these types of systems feature:

> **Concurrency**
> At the least, the concurrency between the system and its environment must be taken into account.
>
> **Time requirements**
> These requirements concern both their input rate and their input/output response time.
>
> **Determinism**
> The outputs of such a system are entirely determined by their input values and by the occurrence times of these inputs.
>
> **Reliability**
> It is commonplace to say that errors in reactive systems can have dramatic consequences, involving human lives and huge amount[sic] of money.
>
> — Nicolas Halbwachs *Synchronous programming of reactive systems*

Two terms, often applied to reactive systems, are avoided:

Embedded
The notion of an embedded system lies with how the system is perceived by its user. The promise of software is to be able to use general purpose hardware to create special purpose machines. A system which can transform its function by executing a different program is perceived as computer-based. An embedded system performs a single specialized function despite using a computer as part of its implementation. A user may know that the operations of an embedded system are performed by a computer, but his expectation when using the system is that it is a specialized, single purpose machine. Common parlance associates embedded systems with being resource constrained. However, modern computing hardware advances blur the distinction of resource constraints and computers exists which span a wide range of resource availability to meet an equally wide range of system requirements. Because the term *embedded* does not shed much light on the manner in which the system performs, this book does not use the term.

Real-time
A system either does or does not have requirements which specify its response timing. Failure to meet specified timing constraints constitutes a system defect regardless of whether there is any external manifestation or consequence of the failure. The existence of time response requirements does not describe effectively the manner in which a system is designed to meet those requirements. There is no single design or programming approach that is necessitated by response time requirements. The terms *hard real time* or *soft real time* provide no additional clarity. It seems, colloquially, that *hard* time requirements are meant to be taken seriously, having possible grievous consequences if not met. There is sometimes an implication that the timing requirements may pose difficulties for the implementation to achieve, conflating the ideas of strict and difficult. *Soft* timing requirements usually equate to, "not too long as to annoy the human operating the system."

Target Readers

This book is primarily intended for:

- Experienced developers looking for new system building techniques.
- Engineers seeking a method of building software systems the result of which is easier to understand and reason about.
- Software developers interested in software engineering concepts which are validated by an accompanying implementation of the concepts.

In the end, we must conclude this is *not* a beginners book. Albeit there is a large amount of *explanation* in the book, there is no deliberate attempt at being a tutorial. There are, for example, no exercises or other comprehension tests. The explanation is intended to describe the design concepts and the accompanying implementation demonstrates the concept in practice. Basic knowledge of programming, particularly in "C", is assumed. There are many good sources of tutorial material on the Internet that are not replicated here. This is not to say that a beginner would not benefit from reading the book. Rather beginners are advised that it may be necessary to search other sources for some background information to which they may not have been exposed previously.

Overview

This book is divided into five parts. The first three parts contain the main exposition of the subject matter. The remaining two parts cover common code used elsewhere and supplemental information for the benefit of readers.

- Part I, *Tap Root*, describes the core systems programming that is required to execute a program and have that program interact with its environment.
- Part II, *Baton*, describes the execution model for unprivileged processing.
- Part III, *Canopy*, describes developing a system using the components from Parts I and II.
- Part IV, *Supporting Code*, contains chapters describing common code sequences used elsewhere in the book.
- Part V, *Supplemental Materials*, is a collection of terms, appendices and other information which is referenced elsewhere in the book.

What is the bottom?

The subtitle of this book is, *A Microcontroller from the Bottom Up*, so what constitutes "bottom" must be defined. There are many deep wells of knowledge and expertise in computing electronics. It ranges from the physics of semiconductors, to the design of instruction sets, and to the amazingly complex machines used to manufacture semiconductors. Since our primary interest in this book is software and, standing firmly on the shoulders of others, the starting point is taken as a complete microcontroller hardware system, but without any software. This starting point is quite elevated in the computing food chain. There are many important aspects of how microcontroller hardware is made into working computer systems that are not considered. It is assumed that there is some technique to install code on the microcontroller and the software execution can be monitored and controlled.

The criteria for choosing a platform for demonstrating the implementation are:

- A relative modern computing architecture whose processor core supports some form of execution constraints.
- The availability of inexpensive hardware development boards.
- The availability of language tools, debuggers, and general development ecosystem.

There are many choices which meet these criteria. Components from SparkFun® have been chosen. The target implementation is the Artemis MicroMod processor board installed on a suitable carrier board. The processor is an ARM® Cortex-M4 based SOC manufactured by Ambiq®.

Some readers will unfortunately conclude that since the target implementation is a microcontroller and their interests are in larger, more capable machines that this book has little to offer them. Since the intent is to have a functioning body of software, there is substantial, unavoidable SOC specific implementation code in the book. The choice of using a microcontroller avoids many of the complexities of larger computing environments so that the focus can remain on the essentials of the engineering. However, most of the microcontroller specific details are contained in Part I and later parts of the book make little or no reference to target specific details. That is one of the first separations of concern we make, namely, an adamant distinction between the application logic and the specific computing technology used to implement that logic. To that end, Part II gives an implementation of the basic execution scheme both for the Artemis processor board and the POSIX environment. As title indicates, the book progresses from specific code to more abstract models of application logic in an effort to both scale development efforts to build complex, yet understandable, systems and to demonstrate clearly how those techniques are composed from fundamental constructs.

What is the top?

The goal is to be able to build complex software systems which are understandable and sustainable. That will be accomplished by decomposing the problem in to cohesive domains based on subject-matter. The domains are individually analyzed. The analysis is captured in a model of the subject matter of the domain. That model is then *translated* into the system implementation. The modeling artifacts provide two essential functions:

a. A means to precisely specify the problem being solved. This implies that understanding about the system is *not* derived from the code which implements it, but rather by "executing" the model.

b. A direct path from the model to an implementation. Model artifacts must contribute directly to deriving the system implementation.

A system is then composed of a set of highly cohesive domains interacting with each other and the environment by means of *bridges*. Bridging is the activity required provide the mapping between the semantics of the various subject matter domains.

In the book Models to Code, the approach started at modeling and demonstrated how code could be directly derived from the models. In this book, the approach is in the opposite direction to show how models help conquer the problems of scale in understanding and building complex systems.

There Will Be Code

And a lot of it. This book is also a literate program. Code and description are combined in the document to facilitate understanding of the program logic. The source code is *tangled* (*i.e.* separated) from the document to present to the compiler. As suits our target platform, the programming language is "C". There is a simple syntax for denoting program parts within the document and the details are presented in an appendix.

Literate programming allows a program to be presented to the reader in an order which is different than that demanded by the compiler. It allows the presentation of design concepts from top down, bottom up, or inside out as it appropriate to the subject matter.

How to Read this Document

Your author can assure you that this book was not written sequentially starting at page one. There was much skipping around to develop an organization for the presentation. There is no expectation that readers will start at page one and read the book sequentially. Skipping around is encouraged.

Part I of the book necessarily contains a substantial amount of processor specific systems programming and discussion of how system memory is handled. The *Crossing the Divide* chapter is central to Part I. This chapter defines the mechanisms by which

the system interacts with the environment and how that interaction is brought into the system for further actions. Reading this chapter is central to understanding subsequent parts of the book.

Part II is devoted to the model of execution and the run time code used to implement it. The approach to the execution model is distinctive in that all decisions about how control is sequenced and logic is synchronized and all policies regarding how data is managed are factored into the run time execution code. Application code has limited ways in which it can affect execution sequencing and manage its data and, consequently, makes no decisions as to which body of code is next executed. The execution model yields deterministic behavior that is only dependent upon the timing of happenings in the system environment. Data is dealt with transactionally and its consistency is checked to insure the semantic rules of the application are maintained throughout a program's execution.

Part III show how application logic is separated from implementation technology by translating semi-formal models into running programs. Essential to building complex systems is the ability to compose larger software pieces together and Part III defines what those pieces are and how they are wired together to achieve the purpose of the application.

Move Along — Nothing New Here

It is important to emphasize that this book does not present any novel computer science concepts. The ideas in this book are an application of the fundamentals of Computer Science which have been around for decades. The more abstract concepts owe much to the work of Tom DeMarco, Stephen Mellor and Paul Ward, and, in particular, Sally Shlaer and Stephen Mellor. The only unique content in this book is the particular implementations and organizations that apply the concepts, which themselves were worked out long ago, to the engineering effort of building robust and sustainable systems of known functionality.

Part I

Tap Root

This part of the book shows both the core systems programming necessary to get any program to run on the target and the fundamental mechanism whereby changes in the environment can be made known to the application logic. These ideas are related in the sense that substantial systems programming is required to set up our intended execution environment. That environment separates privileged from unprivileged execution and uses the Memory Protection Unit to enforce the separation of the two modes of execution. Separating execution privilege requires mechanisms to allow the two execution types to interact in a highly controlled fashion. That interaction is governed by the processor architecture and requires yet more systems programming to implement. The scheme for interacting with the environment includes both mechanisms to control peripheral devices and an implementation pattern for device interactions that provides for interrupt level processing to inform unprivileged execution of significant happenings in the environment. This mechanism is fundamental to the design scheme for handling the currency implicit in the environment and is accomplished using processor interrupts to preempt unprivileged execution. The manner in which the preemption is scheduled is entirely constructed from the processor's priority scheme used for exception handling. A critical part of the scheme allows interrupt handling functions to post continuation requests for additional processing to be performed by unprivileged code.

Chapter 2

The Path to `main`

This chapter shows the code necessary to bring the processor to a state where it can execute the `main()` function. There are many ways to accomplish start up. Most vendors supply sufficient start up code to build a program. Usually vendor supplied code is just sufficient to be able to compile and link a program. You will want to do substantially better.

Executing from power up

Getting a computer to do useful work from power up is a surprising difficult undertaking. Desktop computers running an operating system go through many phases of initialization just to get to the point where you can run any program at all. When writing programs for an established operating system, the tedious aspects of initializing the system to run a program have already been done for you. The language compilers and other tools have already been configured to produce an executable that the operating system can load and run. After all, easily running a set of application programs is an essential aspect of an operating system.

A microcontroller usually only runs one program at at time. Coming from the bottom up, you are faced with the task of determining out how to produce an executable which is acceptable to the processor. Starting up a microcontroller is simpler, but requires detailed knowledge in four particular areas. This is an unfortunately common situation of needing to know a lot in order to do anything at all.

1. The processor architecture insists upon a certain arrangement of data values in memory at start up time and has a precise mechanism it uses to begin execution.
2. The "C" compiler must be directed to place the required code and data in memory, organized in the manner the processor requires. The start up code could be written in assembly language where complete control is possible. It is more convenient to have the source code in one language, even if that means resorting to *inline assembly* language on rare occasions.
3. The linker must be directed to locate the startup code and data at the place in memory where the processor expects to find it. The linker must also be directed where to place other code and data that has not yet been written.
4. The system specific initialization must be done. This usually involves at least setting up clock sources and frequencies.

For ARM Cortex-M4 processors, power up is one of the conditions where the Reset exception is executed and the mechanism for executing any exception starts with the exception vector table. When the processor powers up, it initializes the value of the stack pointer to the first element of the exception vector table. The processor then loads the program counter with the value of the second element. The second element in the vector table is the pointer to the Reset exception handler function. The internal registers and state of the processor are set to known values. For an SOC, there can be several forms of reset. Some peripheral device registers are initialized to reset values while others may preserve their values when the reset does not originate from power up. Once the reset state is established, the microcontroller does what all computers do — fetch the next instruction, as given by the Program Counter, and execute it.

Graphically, a generic Cortex-M4 vector table appears as follows. [1]

[1] ARM DUI 0553A, ID121610, p2-24

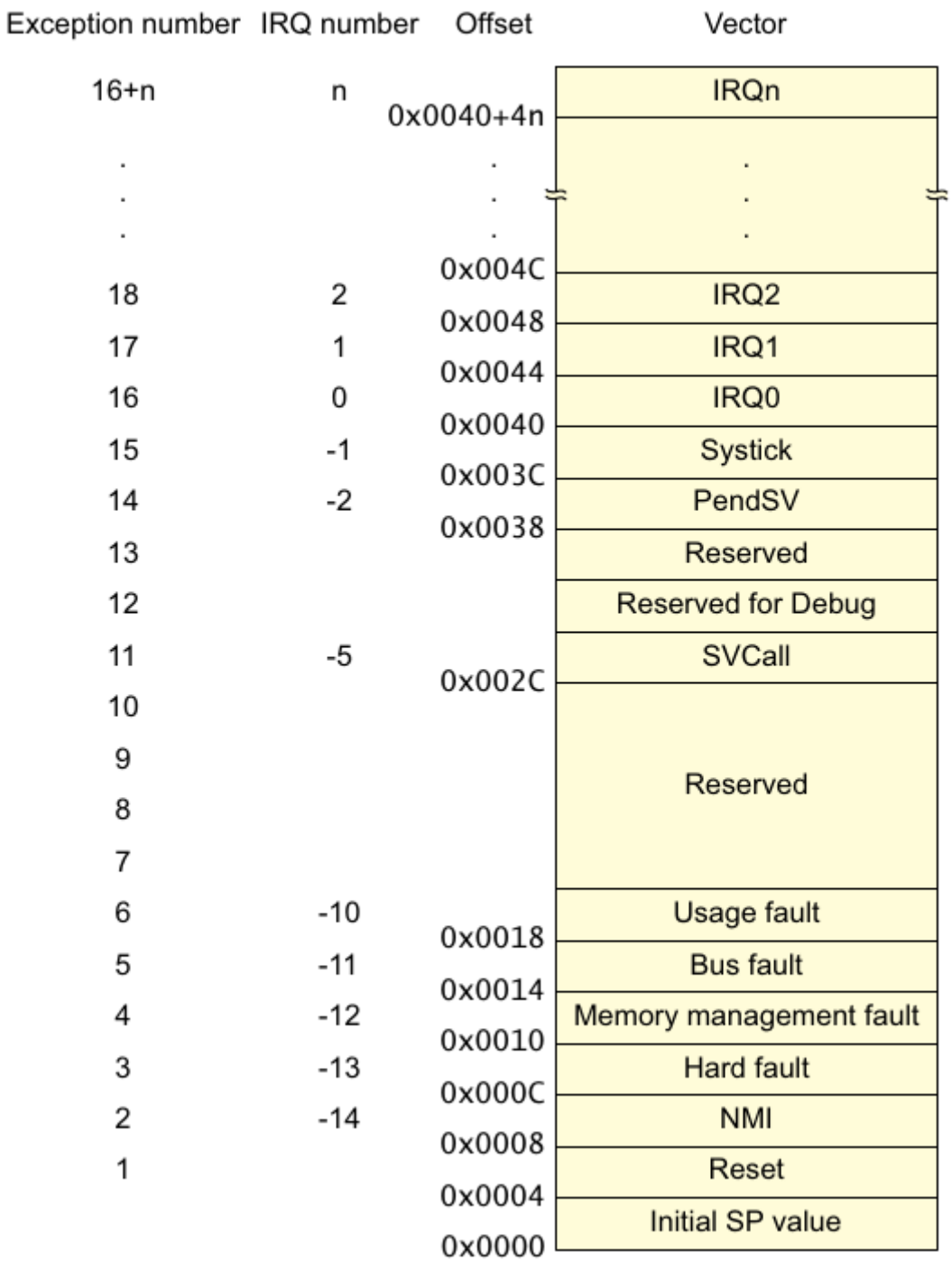

Figure 2.1: Cortex-M Vector Table

The vector table is specialized by the Apollo 3 SOC and appears as follows. [2]

[2] Apollo 3 MCU Datasheet, Doc. ID: DS-A3-0p15p0, p.71

Exception Number	IRQ Number	Offset	Vector	Peripheral/Description
255	239	0x03FC	IRQ239	
.		.	.	
.		.	.	
.		.	.	
.		0x00C0		
.		0x00BC	IRQ31	Clock Control
.		0x009C	IRQ23-30	Stimer Compare[0:7]
.		0x0098	IRQ22	Stimer Capture/Overflow
.		0x0094	IRQ21	SW INT
.		0x0090	IRQ20	MSPI
.		0x008C	IRQ19	PDM
.		0x0088	IRQ18	ADC
.		0x0084	IRQ17	SCARD
.		0x0080	IRQ16	UART1
.		0x007C	IRQ15	UART0
.		0x0078	IRQ14	Counter/Timers
.		0x0074	IRQ13	GPIO
.		0x0070	IRQ12	BLE
.		0x006C	IRQ11	I^2C/SPI Master 5
.		0x0068	IRQ10	I^2C/SPI Master 4
.		0x0064	IRQ9	I^2C/SPI Master 3
.		0x0060	IRQ8	I^2C/SPI Master 2
.		0x005C	IRQ7	I^2C/SPI Master 1
.		0x0058	IRQ6	I^2C/SPI Master 0
.		0x0054	IRQ5	I^2C/SPI Slave Register Access
.		0x0050	IRQ4	I^2C/SPI Slave
.		0x004C	IRQ3	Voltage Comparator
18	2	0x0048	IRQ2	RTC
17	1	0x0044	IRQ1	Watchdog Timer
16	0	0x0040	IRQ0	Brownout Detection
15	-1	0x003C	Systick	
14	-2	0x0038	PendSV	
13			Reserved	
12			Reserved for Debug	
11	-5	0x002C	SVCall	
10			Reserved	
9				
8				
7				
6	-10	0x0018	Usage Fault	
5	-11	0x0014	Bus Fault	
4	-12	0x0010	Memory management Fault	
3	-13	0x000C	Hard fault	
2	-14	0x0008	NMI	Unused
1		0x0004	Reset	
		0x0000	Initial SP value	

Figure 2.2: Ambiq Apollo 3 Vector Table

Note that the first 16 exception vectors are the same as required by the Cortex-M4 architecture. The remaining 32 exception vectors are specific to the peripherals of the Apollo 3.

Note

As shown in the previous figure, ARM makes a distinction between the *Exception number* and the *IRQ number*. Exception numbers have a direct association to elements of the vector table. An IRQ number is used as an index into the vector table in some contexts, *e.g.* CMSIS functions use IRQ numbers in its operations. For now, just remember the distinction and endeavor to keep the terms straight.

Another goal is to create a means of start up that can be used repeatedly when building programs for the device. Even if usually only running one program at a time is running, it is necessary to build conveniently multiple programs.

> I consider building an application as building and running a long series of test applications — the last one of which is pronounced, "`finished`" [a].
>
> [a] Whatever finished means in software.

Vector table

Except for the first element being a pointer to memory suitable for holding the stack (*e.g.* RAM memory), the exception vector table is just an array of pointers to functions. Computer memory doesn't have any associated type — data types are a programming language construct.

There is one further complication which is sometimes overlooked and can cause confusion. It has to do with how the processor encodes instruction set selection in a function pointer. ARM processors have had a 32-bit ARM instruction set and sometimes a more densely encoded 16-bit instruction set. The 16-bit instruction set is called, *Thumb*, and comes in two variations, Thumb-1 and Thumb-2[3]. Cortex-M4 processors only execute the Thumb-2 instruction set. ARM processors that execute the 32-bit instruction set can switch instruction encodings between the 32-bit instructions and the 16-bit instructions. The difference is encoded in the least significant bit (LSB) of the function pointer value. This bit is not used for actually addressing an instruction since instructions must be aligned to even addresses. To detect if 32-bit instructions got mixed into a Thumb-2 processor, the function pointers in the exception vector table all have the LSB set. A raw dump of memory would show the pointer addresses as all odd numbered, even though the instructions for a functions definitely do *not* start on an odd address. In certain situations such as this one, bits are precious and are used to overload the meaning of a memory address.

The goal is to arrange a sequence of values where each value is 32-bits and must be specified as the appropriate function prototype. It is convenient to define a `typedef` for the function prototype.

```
<<startup_apollo3: data type declarations>>=
typedef void (*ExceptionHandler)(void) ;
```

In words, an exception handler type is a pointer to a function which takes no arguments and returns no value. That's in keeping with nature of exception level execution. Exceptions arise from either the internal operations of the processor or by signals from outside the processor, temporarily suspend computation to run the exception handler, and returning to the interrupted location to resume the suspended computation. There really is no way to pass an argument and no call site which is expecting a returned value[4]. That doesn't mean that exception handlers can have no effect on the system. Since exception handlers are just running code, they can read and write memory and that is all the mechanism needed to have an exception handler affect subsequent processing.

Notice there is no special syntax to tell the compiler that the function is an exception handling function. Many processor architectures have specific requirements as to how exception handling code must interface to the system and compilers often support annotation of exception handlers so they may generate the correct code. On the Cortex-M4 architecture, the hardware arranges the entry to an exception handler according to the Procedure Call Standard (AAPCS). which is defined for the processor

[3] The whole truth is more detailed. Instruction set variations exist because of optional core components such as the FPU. Here you are concerned with the ARMv7E-M instruction set, including the optional single precision floating-point instructions.

[4] There is one exception, to this circumstance, `SVCall`, and that is discussed later.

architecture. The net effect is that exception handlers look just like ordinary functions at the binary level and there is no need for special compiler annotations.

The processor architecture requires data values in the layout shown in the previous figure. That layout of memory is an array which holds function pointers to our exception handlers. There are two broad types of exception handlers:

1. Those defined by the processor architecture. These are defined by ARM and are the same for all Cortex-M4 cores.
2. Those associated with peripheral interrupts. These are defined by the chip manufacturer and differ from chip to chip.

CMSIS has a number of pre-processor definitions that can be used here.

> In practice one does not truly program in the "C" language but rather in some dialect of "C". All compilers have extensions and abilities beyond what the standard prescribes. It is possible to use the "C" preprocessor to *macro-ize* a common set of compiler features. That is *not* done here on the basis that it just adds to the cognitive load which is already high enough. Since GCC is so readily available, its dialect is used exclusively.

The exception vector table is defined as:

```
<<startup_apollo3: exception table definition>>=
ExceptionHandler const
__Vectors[16 + 32] __VECTOR_TABLE_ATTRIBUTE  = {        // ❶
    <<startup_apollo3: processor exceptions>>
    <<startup_apollo3: peripheral interrupts>>
} ;
```

❶ There's a lot of information in this definition:

- The exception vector table is in reality an initialized array variable named `__Vectors`.
- The vector table size is defined to fit only the exceptions defined by the system (16) and those defined by the Apollo 3 as interrupts (32). This limit needs to be in place since the Apollo 3 defines a patch area for the Bluetooth controller which is expected to follow immediately after the IRQ handlers.
- The variable has a `const` qualifier indicating writes to the variable are not allowed. The vector table is placed in read only memory.
- Finally, there are attributes associated with the `__Vectors` variable. `__VECTOR_TABLE_ATTRIBUTE` is a CMSIS defined macro to tell the compiler that the `__Vectors` variable is actually used (since it is not referenced anywhere in the code) and it should be placed in a linker section name, `.vectors` (so it can be placed precisely using a linker script). The CMSIS definition of the macro accommodates the different syntax used by different compilers to accomplish the variable attribute designation.

Before getting to the exception handling functions, you need to decide what to do about a stack. Having a stack is essential to calling functions since it holds parameter and return information. The stack must be placed in read/write memory. That is almost always supplied by RAM which is part of the SOC. It is possible and sometimes desirable to have several stacks and the Cortex-M architecture defines multiple stack pointer registers. Before considering those more complex scenarios, start with the simple case of providing a Main Stack Pointer (MSP).

```
<<startup_apollo3: processor exceptions>>=
(ExceptionHandler)&__main_stack_top__,              // ❶
```

❶ The cast is necessary to inform the compiler that the address of the stack is to be typed the same as the other elements of the array. Alternatively, you could have defined the vector table to be a structure whose first element is a data pointer (rather than a function pointer), but that is an additional complication avoided with the cast. The start up code doesn't access the vector table so being that strictly typed doesn't grant us much advantage.

So where did the symbol `__main_stack_top__` come from and why is its address taken? The system stack is discussed below when memory initialization is discussed. For now, the `__main_stack_top__` symbol is created by the linker and has been given a distinctive name that is unlikely to collide with any other names in the program (the symbol name is globally known). The symbol value is the address of the top of the stack. Since the stack grows downward in memory addresses (known as a *full descending stack*), the initial value is set to the largest memory address allocated to the stack[5].

A header file is needed to define the standard type names for integers of fixed size.

```
<<startup_apollo3: include files>>=
#include <stdint.h>
```

The exception handler names for the system exceptions defined by the processor architecture are standardized by CMSIS.

```
<<startup_apollo3: processor exceptions>>=
                            // Exception number
Reset_Handler,              // 1
NMI_Handler,                // 2   Unused on the Apollo 3.
HardFault_Handler,          // 3
MemManage_Handler,          // 4
BusFault_Handler,           // 5
UsageFault_Handler,         // 6
Default_Handler,            // 7   Exceptions number 7 - 10 are reserved.
Default_Handler,            // 8
Default_Handler,            // 9
Default_Handler,            // 10
SVC_Handler,                // 11
DebugMon_Handler,           // 12
Default_Handler,            // 13   Reserved
PendSV_Handler,             // 14
SysTick_Handler,            // 15
```

The usual naming convention for SOC specific interrupts is to use the interrupt name given in the SVD file and append `_IRQHandler`.

```
<<startup_apollo3: peripheral interrupts>>=
                                  // IRQ number
BROWNOUT_IRQHandler,              // 0
WDT_IRQHandler,                   // 1
RTC_IRQHandler,                   // 2
VCOMP_IRQHandler,                 // 3
IOSLAVE_IRQHandler,               // 4
IOSLAVEACC_IRQHandler,            // 5
IOMSTR0_IRQHandler,               // 6
IOMSTR1_IRQHandler,               // 7
IOMSTR2_IRQHandler,               // 8
IOMSTR3_IRQHandler,               // 9
IOMSTR4_IRQHandler,               // 10
IOMSTR5_IRQHandler,               // 11
BLE_IRQHandler,                   // 12
GPIO_IRQHandler,                  // 13
CTIMER_IRQHandler,                // 14
UART0_IRQHandler,                 // 15
UART1_IRQHandler,                 // 16
SCARD_IRQHandler,                 // 17
ADC_IRQHandler,                   // 18
PDM_IRQHandler,                   // 19
MSPI0_IRQHandler,                 // 20
SOFTWARE0_IRQHandler,             // 21
STIMER_IRQHandler,                // 22
STIMER_CMPR0_IRQHandler,          // 23
```

[5] Technically, the value is one past the last byte allocated to the stack. The stack pointer is always pre-decremented before it is used to store a value on the stack

```
STIMER_CMPR1_IRQHandler,            // 24
STIMER_CMPR2_IRQHandler,            // 25
STIMER_CMPR3_IRQHandler,            // 26
STIMER_CMPR4_IRQHandler,            // 27
STIMER_CMPR5_IRQHandler,            // 28
STIMER_CMPR6_IRQHandler,            // 29
STIMER_CMPR7_IRQHandler,            // 30
CLKGEN_IRQHandler,                  // 31
                                    // ❶
```

❶ The Apollo 3 chip uses a patch table that follows immediately after the vector table for 16 entries. There is little documentation of this other than its appearance in their start up files and it is apparently used in conjunction with the Bluetooth Low Energy (BLE) module.

Exception vector functions

At this point, an array variable has been defined that is to be the exception vector table and names given to the functions that form the initializer for the array. But now you need to make an important decision about how to supply the code for the exception handling functions.

The simple approach is to open up a file and start typing in the functions whose names are in the vector table array. That approach works fine, but if you think a little further ahead, you realize that it is not very flexible. At this point in time, you don't know what code to put into the function. You could stub out the functions, but then every time you did decide what an exception handler should do, you would have to edit that file. Every different program would have its own file with the exception handling function in place. You would like to use this arrangement of system level code for any program intended for the Apollo 3. For that to happen, you must be able to do two things.

1. Supply your own function without having to edit any system level files. You would like simply to define a function whose name is the same as an exception handler, *e.g.* `GPIO_IRQHandler`, and have that function used in the vector table.
2. Supply a default handler function for any exceptions not used in a program.

Needless to say, this problem has been solved and it is accomplished by giving information to the linker about how to resolve symbolic information.

In the first case, the function is declared to have *weak* linkage. Normal function definitions are considered by the linker as *strong* and attempting to link multiple object files which have strong symbols of the same name is an error. The linker would have no basis for choosing the correct one. However, a weak symbol tells the linker to use a strong symbol of the same name if one is available and otherwise use the weak symbol.

The second case is handled by defining an *alias*. This directive informs the linker to use a different value for a symbol. The combination of declaring a function as both weak and giving it an alias achieves our goal of being able to override easily a default implementation of an exception handler.

Not every exception handler needs the flexibility to be substituted. The `Default_Handler` is the default function for the aliases and must be defined in this file. It is not possible to alias a function to an external or undefined function. That's just too much indirection. The `Reset_Handler` is can be defined so that is does the correct operations irrespective of the specifics of an application. Swapping out what to do at reset is not common and if necessary requires changes to the `Reset_Handler` definition.

Neither of these exception handlers ever return. They represent the extremes of the processing spectrum. The `Reset_Handler` is entered upon power up or reset, transfers control to our application program, and there is no call site to which to return. The `Default_Handler` is entered if an exception is taken and no appropriate handler has been supplied, indicating something is terribly wrong, and it's not possible to go forward with any further program execution. Both functions are shown below.

Modern "C" gives us a standard way to declare that a function does not return, but a header file is needed to make that convenient.

```
<<startup_apollo3: include files>>=
#include <stdnoreturn.h>
```

Since this is “C”, there is a long series of declarations to introduce the names of the exception handlers along with the declarations of symbol strength and aliased behavior.

```
<<startup_apollo3: external declarations>>=
void NMI_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void HardFault_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void MemManage_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void BusFault_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void UsageFault_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void SVC_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void DebugMon_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void PendSV_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void SysTick_Handler(void) __attribute__((weak, alias("Default_Handler"))) ;
void BROWNOUT_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void WDT_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void RTC_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void VCOMP_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void IOSLAVE_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void IOSLAVEACC_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void IOMSTR0_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void IOMSTR1_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void IOMSTR2_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void IOMSTR3_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void IOMSTR4_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void IOMSTR5_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void BLE_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void GPIO_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void CTIMER_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void UART0_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void UART1_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void SCARD_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void ADC_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void PDM_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void MSPI0_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void SOFTWARE0_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_CMPR0_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_CMPR1_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_CMPR2_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_CMPR3_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_CMPR4_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_CMPR5_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_CMPR6_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void STIMER_CMPR7_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
void CLKGEN_IRQHandler(void) __attribute__((weak, alias("Default_Handler"))) ;
```

Hitting Rock Bottom

It may seem strange to consider at this time what happens when the software gets to a spot where it does not know how to proceed. This is the classic *this should never happen* case, which during development, all too frequently does actually happen. It’s desirable to have a single function to call when the software doesn’t know what else to do. That function must be used judiciously, but is a key part of always maintaining control of the processor execution. The name chosen for the function, which is derived from a venerable, historic name, is `SystemAbend()`.

So what is the last act of a completely confounded microcontroller? Reset. Strange as it may sound, sometimes resetting the processor and starting all over again is the only recovery available. But, if a debugger is connected, popping back to the debugger is a great help to the developer. Fortunately, the Cortex-M architecture has the capability to determine if a debugger is enabled and an instruction to break out to it.

```
<<system_apollo3: external function declarations>>=
extern noreturn void SystemAbend(void) ;
```

```
<<system_apollo3: external functions>>=
__WEAK
noreturn void
SystemAbend(void)
{
    if (stwd_debug_enabled() || stwd_debugmon_enabled()) {   // ❶
        __BKPT(0) ;                                          // ❷
    }

    NVIC_SystemReset() ;                                     // ❸
}
```

❶ The test of whether Halting debug or the DebugMon exception are enabled before executing a `BKPT` instruction may seem superfluous. The object of the test is to prevent causing a Hard Fault while trying to terminate. If debugging is enabled or the DebugMon exception is enabled then `BKPT` would cause a *debugging event* to be generated. However, if neither of these conditions is enabled, as might be the case in a deployed system, the debugging event would cause the generation of a Hard Fault. You want to be able to invoke `SystemAbend()` even in the Hard Fault handler. Creating a fault in the Hard Fault handler causes lockup, a situation to avoid since it complicates the diagnosis of any problem where the running program itself causes the lockup.

❷ The zero argument is actually an immediate operand in the assembly language instruction. The choice of number is somewhat arbitrary, but some break point numbers are used for *semi-hosting*, a older arrangement for obtaining debugging output which is not used here. ARM recommends *not* using 0xab as the immediate value since this is the value conventionally used for semi-hosting.

❸ This is a Core CMSIS function which resets the processor through the NVIC. The CMSIS implementation insures the function never returns.

To access Core CMSIS definitions and other chip definitions, a header file for the Apollo 3 is included.

```
<<startup_apollo3: include files>>=
#include "apollo3.h"
```

This design supplements the Core CMSIS definitions with some simple functions that manipulate the controls bits of the system registers. Named functions are used to manipulate hardware registers.

```
<<startup_apollo3: include files>>=
#include "bit_twiddle.h"
#include "sys_twiddle.h"
```

The sys_twiddle and bit_twiddle functions are discussed in their own chapters.

Default exception handler

Most vendor supplied default exception handlers just go into a tight, infinite loop. In that case, when things go terribly wrong the system hangs. If you happen to have a watchdog timer going, it would probably reset the system. There are many options for what to do in the default handler. For a more robust system which supports easier diagnosis of fault problems, you should leave "`bread crumbs`", *i.e.* copies of important status registers that determine what went wrong. Of course, if you leave a trail, you also must provide a means to examine what was left to diagnose the problem. Handling the boundaries conditions of a system going up or down, *e.g.* when its battery runs out of energy, is a broad topic deferred until later.

Starting at the bottom, it is difficult to know exactly what must be done for the exception handling. A means of adding in better exception handling later is needed. That means gathering information when the exception happens so that it may be used to diagnose the exception later. Similar to what was done with the default exception handler itself, more substantial processing can be provided by overriding functions. In particular two features are needed:

1. Gather the exception information required for better handling. Namely, access to the stack frame created when the exception occurred, the exception return information, and the exception number which is running are needed.

2. Forward the handling of missing exception handlers to a function that can be overridden for the specifics of your particular system. Missing exception handlers are usually an indication of a problem building the program where the intended handler does not get included to override the *weak* default handler.

Because in this case a generic solution is desired, some assembly language is required.. The vast majority of Cortex-M programs do not require any assembly language. The design of the processor and the AAPCS are coordinated to remove the need for *glue* assembly language just to write an exception handler. But inevitably, systems programming near the bottom requires a small amount of assembly to access system specific information which can be used by compiled code for analysis.

For readers unfamiliar with ARM Thumb-2 assembly language, the techniques used to accomplish the above goals can be safely skipped. For others, this is the time to roll up your sleeves and dive in.

The design goal is to make the `Default_Handler` a wrapper around a general function for missing exception handlers. The `Default_Handler` gathers the necessary information and invokes `SystemMissingHandler()`. The definition of `SystemMissingHandler()` is given below.

Three arguments are provided to `SystemMissingHandler()`.

1. A pointer to the exception stack frame.
2. The exception return value.
3. The value of the IPSR status register which holds the exception number of the currently executing exception.

The memory layout of an exception stack frame for a Cortex-M4 is captured by the following “C” structure definition.

```
<<system_apollo3: data type declarations>>=
typedef struct __attribute__((packed)) {
    uint32_t r0 ;
    uint32_t r1 ;
    uint32_t r2 ;
    uint32_t r3 ;
    uint32_t ip ;
    uint32_t lr ;
    uint32_t pc ;
    uint32_t xpsr ;
#       if defined(__FPU_USED) && (__FPU_USED == 1)
    float s0 ;
    float s1 ;
    float s2 ;
    float s3 ;
    float s4 ;
    float s5 ;
    float s6 ;
    float s7 ;
    float s8 ;
    float s9 ;
    float s10 ;
    float s11 ;
    float s12 ;
    float s13 ;
    float s14 ;
    float s15 ;
    uint32_t fpscr ;
#       endif /* __FPU_USED */
} ExceptionFrame ;
```

There are two items to note here:

1. If the FPU is not being used, there is no possibility that the stack frame contains the saved floating point registers.
2. Even if the FPU is used, the exception entry may *not* have pushed the registers. There are control settings in the `FPCCR` register which determine the stacking of FPU registers during exception entry.

To adhere to the AAPCS when invoking the missing exception handler, `exc_frame` must be placed in **R0**, `exc_return` in **R1**, and `ipsr` in **R2**. With the arguments in place, the control flow is a normal subroutine call to `SystemMissingHandler()` followed by an exception return. The exception return prompts the processor to restore the state information stacked during exception entry to resume processing where the exception took place. Note in this case, there is no exception return. The code for `SystemMissingHandler()` invokes `SystemAbend()`.

There is one other consideration to be handled. The processor builds an exception frame on whichever stack is in use when the exception occurs. The exception handling code itself always uses the MSP, but if the PSP was in use when the exception occurred, then it is used for the exception stack frame. Fortunately, bit 2 of the exception return value ,which is in the link register, tells which stack pointer is to be used when the exception returns and, consequently, which stack pointer contains the exception stack frame.

The following assembly language implements the design.

```
1  <<startup_apollo3: static functions>>=
2  __attribute__((naked))
3  static void
4  Default_Handler(void)
5  {
6      __asm__ volatile
7      (
8          "tst    lr,#4                           \n\t"
9          "ite    eq                              \n\t"
10         "mrseq  r0,msp                          \n\t"
11         "mrsne  r0,psp                          \n\t"
12         "mov    r1,lr                           \n\t"
13         "mrs    r2,ipsr                         \n\t"
14         "push   {lr}                            \n\t"
15         "bl     SystemMissingHandler            \n\t"
16         "pop    {pc}                            \n"
17     ) ;
18 }
```

Line 2
The `naked` attribute requests the compiler *not* to include the usual function prologue and epilogue instructions. This allows us to provide a pure assembly language function in the guise of an ordinary "C" function without unwanted compiler generated instructions.

Line 8
The test of bit 2 in the link register determines which stack was in use when the exception occurred. If the bit is clear, then the MSP was used to stack the exception state. If the bit is set, then the PSP was used for the exception state information.

Lines 9-11
This is an example of conditional instruction execution that is part of the ARM architecture. Only one of the two instructions on lines 10 and 11 actually execute. The other instruction does not execute because its condition code does not match the condition of the `ite` instruction. This saves explicit branching when the number of instructions around the branch location is small.

Line 12
The link register holds the value of exception return. It is copied to **R1** to become the second argument.

Line 13
The IPSR is one view of the extended processor status register and the `MRS` instruction is used to access it. The ISPR value is placed in **R2** to become the third argument.

Line 14
The value of the link register must be saved in preparation for invoking a function.

Line 15
Invoking the function is accomplished by executing a `BL` instruction which stores the return address (*i.e.* the value of the PC after the `BL` instruction) into the link register and jumps to the beginning of the function.

Line 16
Exception return is accomplished by popping the value of the EXC_RETURN (which is sitting on the stack where it was saved in line 14) directly into the program counter.

The subject of handling exception is further discussed in a subsequent chapter of this book.

Reset Exception Handler

Power up and reset conditions result in a Reset exception and vector table directs the processor to execute the Reset exception handler.

```
<<startup_apollo3: forward references>>=
noreturn void Reset_Handler(void) ;
```

The goal here is to prepare the system to execute a program beginning at `main()`. This involves the following tasks:

1. Get the hardware into the state required execute a program.
2. Initialize read/write memory to the way the “C” language standard requires and that supports our running program.
3. Initialize the exception priorities.
4. Initialize the memory protection.
5. Switch to the Process Stack and disable privileged execution.
6. Transfer control to `main()`.

```
<<startup_apollo3: external functions>>=
noreturn void
Reset_Handler(void) {
    SystemInit() ;

    __set_MSP((uint32_t)&__main_stack_top__) ;          // ❶
    __set_PSP((uint32_t)&__process_stack_top__) ;

    start_main() ;
}
```

❶ Execution is now on a direct path to `main()` and so the stack pointers can be set to their respective top values. At this place, the processor is working off of the Main Stack.

```
<<start main: external functions>>=
noreturn void
start_main(void)
{
    uint32_t control = __get_CONTROL() ;                // ❶
    control = BTWD_FIELD_SET(control, CONTROL_SPSEL) ;
    control = BTWD_FIELD_SET(control, CONTROL_nPRIV) ;
    __set_CONTROL(control) ;
    __ISB() ;

    main(argc, argv) ;

    abort() ;                                           // ❷
}
```

❶ The core `CONTROL` register is used to select the stack used for Thread mode and to remove privilege execution from the running thread. See the following discussion for the rational of using unprivileged execution.

❷ On the off chance that `main` returns, execution is handed off to the `abort()` function.

Note

It may seem strange to create a separate function for `start_main()` and to place it in a separate file. When the `CONTROL` register is set for unprivileged execution, the next instruction must be fetched from memory which has unprivileged access. At this time, the Memory Protection Unit (MPU) may have been configured by the `SystemInit()` function and unprivileged execution must be separated from the privileged execution of the Reset Handler. This is discussed in a subsequent section.

In a subsequent section, the definition of the `SystemInit()` function is shown. In the next section, the code required to initialize RAM memory is presented. A header file is required to declare the function prototypes.

```
<<startup_apollo3: include files>>=
#include "system_apollo3.h"
```

The `main` function is the standard entry point for a program defined by the "C" language standard. In a conventional operating system environment, a command line shell or other program would feed arguments to `main` which contain the invocation parameters. In a microcontroller environment where only one program is run, there still may be "arguments". The arguments might not come from a command line, but it is handy to sometimes arrange for arguments for a particular build of a system. This gives us the ability to use the arguments to control aspects of the program behavior at run time. For example, an argument could direct the program to be extra verbose about some computation. A subsequent build intended for deployment might leave that argument out.

```
<<startup_apollo3: forward references>>=
extern noreturn void start_main(void) ;
```

The arguments are declared as *weak* global variables so that a particular program may override them.

```
<<start main: main argument definitions>>=
__WEAK int argc ;
__WEAK char *argv[1] ;
```

Initializing RAM Memory at Start Up

The "C" language insures that the values of non-automatic variables which have initializers have their initial values in place when `main()` runs. It also insures that non-automatic variables which do *not* have initializers are set to zero, *i.e.* all bits in the variable are zero. With a conventional operating system, the program is set up to bring in the variable initializers from the disk file which holds the program executable image. This is usually done on demand, *i.e.* when the program actually accesses a variable which was initialized. Zero initialized pages can be allocated by the operating system and zeroed in place.

On a microcontroller there may not be a file system or even any external storage. Only one program is typically executed and there is no mechanism to page in from disk. At power up, RAM values are indeterminate, so the variable initializers must be stored somewhere and copied into RAM. Once the RAM memory is initialized, normal program execution can then read or update the values as needed. The most convenient source of non-volatile memory is the flash memory where our program is stored. So, the data initializers must to be stored in flash and copied to RAM before `main` executes. Using flash memory to store initializers does reduce what is available for program instructions, but in most microcontrollers the amount of flash memory far exceeds the amount of RAM memory.

A similar situation exists for zero initialized memory. Since the value to be stored in RAM is just zero, there is no need to waste flash space storing a potentially large array of zeros. So, the memory locations occupied by uninitialized variables must be programmatically set to zero.

Where does the necessary information to copy memory or zero memory come from? The memory addresses are only known at link time and the specifics will vary from one program to another. The linker is able to supply the needed information. How that is done is covered in a following section. For now, assume that there are some symbols whose values are set by the linker. These

symbols give you all the information needed to write the code to perform the memory initialization. Note that the code and the linker directives are now tightly coupled by the names given to those symbols. The chosen names are conventional and follow a pattern, but the names are arbitrary as long as both the initialization code and the linker directives use the same name to mean the same thing.

Two other memory initialization situations arise:

1. Procedure linkages and automatic variables are stored on the stack. It does not need to be initialized for proper operation since the procedure linkages are always written before they are read and automatic variables have indeterminate values before they are written. However, the stack has a fixed size and it is convenient to initialize the stack to some known, improbable value in order to determine determine the extent of stack usage. By initializing the stack area and then running the system, execution can be halted to examine a memory dump of the stack area. The contrast between the improbable initial memory value and what is written during program execution will give a good measure of the extent of stack usage. There are other, and better ways, to protect the system from stack excursions. The MPU can be used or clever stack placement in memory can catch the stack excursions which a deeply recursive function might trigger. But simply initializing the stack memory to known values is an easy way to get started.

2. It is convenient to have a section of memory set aside which is not initialized unless there has been a power on reset (POR). A processor reset can come from many sources and may even be caused intentionally by the software. It is useful during development to leave a trail of clues about the state of the system before the reset by storing information in a section of memory which is not, unlike ordinary variables, initialized at reset. Of course, should the reason for the reset be power on, *i.e.* a hard, cold start, RAM contents are indeterminate and the trail is lost. But during development, using a special section of memory can be a useful tool to uncover difficult to diagnose errors. Such a memory area is also useful for gathering operational statistics which must survive reset. If an external interface to the memory contents is provided (*e.g.* via some communications interface), it can help diagnose problems in a deployed system. Most modern microcontrollers have a specific peripheral to monitor and control resets and that hardware usually has registers which indicate the cause of the last reset. The Apollo 3 has such a register.

The implementation of the RAM initialization follows directly from the considerations of the four memory segments just discussed.

```
<<system_apollo3: forward references>>=
static void init_ram(void) ;
```

```
<<system_apollo3: static functions>>=
static void
init_ram(void)
{
    <<init_ram: data sections>>
    <<init_ram: bss sections>>
    <<init_ram: stack sections>>
    <<init_ram: no init sections>>
}
```

Transferring Linker Symbols into Code

Since the initialization code depends upon the linker for the information needed to initialize the memory segments, how linker symbols and source code symbols interact must be considered. Symbols in source code and symbols created by the linker have different characteristics. Access to linker generated symbols must account for the fact that, unlike source code variables which have both an address and a value, linker symbols may have no associated value. There is a good description in the `ld` documentation as quoted here:

> Linker scripts symbol declarations, by contrast, create an entry in the symbol table but do not assign any memory to them. Thus they are an address without a value. So for example the linker script definition:
>
> ```
> foo = 1000;
> ```

> creates an entry in the symbol table called `foo` which holds the address of memory location 1000, but nothing special is stored at address 1000. This means that you cannot access the value of a linker script defined symbol — it has no value — all you can do is access the address of a linker script defined symbol.
>
> — GNU ld manual

The simple rule is that linker symbols are just physical addresses. There may be something at that address or not. A convention must be established between the values of a linker symbol and any code which uses the symbol. To make the correspondence of the linker symbol names to the names which appear in the code, external declarations for the linker symbols are placed in a header file.

```
<<startup_apollo3: include files>>=
#include "linker_symbols.h"
```

Copying Variable Initializers to RAM

There are couple of conventional tactics used to get the linker derived information into the memory initialization code.

1. You can define linker symbols which mark the boundary of a section and use those symbols to know the source of variable initializers and the RAM destination of the copy. This is simple and works well if the layout of initializer data is contiguous.
2. You can use the ability of the linker to place values into memory to define a *table* which contains a set of source and destination information. This approach easily handles multiple, non-contiguous initializer sections.

Here the second approach is used. It is only slightly more involved (requires a loop to iterate through the table) and does not require significantly more code to implement. The linker script directs the linker to build a *table* containing the needed descriptive information about the memory section.

That table is constructed such that you can declare an equivalent "C" structure. This is, for example, a common technique used to define the layout of memory-mapped peripheral device registers. In this case, the layout and values are being provided through a linker script. You must declare an equivalent "C" structure to enable the code to retrieve the linker-generated values correctly.

The information required to copy data initializers to RAM appears in "C" source code to be an array of structures of the following type.

Data copy structure declaration

```
<<linker symbols: data type declarations>>=
typedef struct {
    uint32_t const *src ;
    uint32_t *dst ;
    uint32_t nwords ;                    // ❶
} CopySegmentDesc ;
```

❶ In units of 32-bit words, *i.e.* `nwords` is the number of 32-bit words to copy. All subsequent `nwords` members are also in 32-bit word units.

The memory initialization code then uses the values in the linker-generated table to perform the initialization. The net effect is no different than if you had built the table by defining the array directly in our own "C" code. But since the needed values are only known at link time, the linker is enlisted to place the values into memory for us. This must be done carefully, as discussed below, to insure matching the memory layout implied by the `CopySegmentDesc` structure.

If you treat the copy table as an array, you must to know where the array ends. If the number of elements was known, then the end could be calculated. However, in laying out the array using linker commands, the easiest solution is to define two linker symbols which mark the start and end of the implied array of copy segments. Then, using use the fact that pointer arithmetic in "C" is always scaled by the size of the object pointed to, you can increment a pointer through the array until the end is reached.

```
<<init_ram: data sections>>=
CopySegmentDesc const *copy_seg = &__data_copy_table_start__ ;
CopySegmentDesc const *const copy_seg_end = &__data_copy_table_end__ ;    // ❶

for ( ; copy_seg < copy_seg_end ; copy_seg++) {
    uint32_t const *src = copy_seg->src ;
    uint32_t const *const src_end = &src[copy_seg->nwords] ;        // ❷
    uint32_t *dst = copy_seg->dst ;

    while (src < src_end) {
        *dst++ = *src++ ;
    }
}
```

❶ If you prefer to formulate the loop in an array indexing style (as oppose to the scaled pointer arithmetic shown), then it is necessary to know the number of elements in the `copy_table`. That is given by:
`(&__data_copy_table_end__ - &__data_copy_table_start__)`
This works because of the external declarations of the start and end and that pointer subtraction is scaled by the size of the type of object pointed to.

❷ This sets `src_end` to one byte past the last byte of the source block. You are allowed to compute such an address, as long as you don't dereference the `src_end` pointer. As you see, `src_end` is where the iteration through the data source stops.

Zeroing Out Uninitialized Variable Values in RAM

The information required to place zero values into memory is simpler since it does not require a source of data to write a constant value into a memory region.

As before with the initialized data, you create a "C" structure declaration to match the layout of data as setup by the linker commands.

```
<<linker symbols: data type declarations>>=
typedef struct {
    uint32_t *dst ;
    uint32_t nwords ;
} InitSegmentDesc ;
```

The implementation of the code to zero out the memory follows a similar pattern used to copy over the initialized data values.

```
<<init_ram: bss sections>>=
InitSegmentDesc const *zero_seg = &__zero_table_start__ ;
InitSegmentDesc const *const zero_seg_end = &__zero_table_end__ ;

for ( ; zero_seg < zero_seg_end ; zero_seg++) {
    uint32_t *dst = zero_seg->dst ;
    uint32_t *dst_end = &dst[zero_seg->nwords] ;

    while (dst < dst_end) {                                    // ❶
        *dst++ = UINT32_C(0) ;
    }
}
```

❶ Here, standard library functions such as `memset()` are avoided to eliminate any dependency on an external library in the start up code.

Setting the Stack Memory to a Known Value

Before discussing initialization of the stack section, two design decisions must be made:

How many stacks do are to be supported?
The ARMv7-M architecture supports both a Main Stack and a Process Stack. You must supply a Main Stack.

Where in memory is the stack to be located?
There are several strategies which are practiced. The stack, unlike most other memory usage, grows down, *i.e.* smaller stack addresses hold more recent stack data.

For this use case, a two stack arrangement is supported for the following reasons.

- Exception processing always uses the Main Stack. This type of processing tends to be better bounded in terms of its stack usage. It is unlikely to have any recursion or need large amounts of automatic variable space.
- The Process Stack is used by application code and is much less predictable in the amount of memory it requires.
- By separating the Process Stack from the Main Stack, it is unnecessary to add extra space in the Process Stack for the stack space needed by exceptions. Exception nesting is always present since high priority system faults can occur at any time, *i.e.* you must always account for Hard Fault and NMI.
- Separate Main and Process Stacks offer the opportunity to restrict access to the memory based on privilege. This could be used to prevent unprivileged code from accessing the Main Stack.

The considerations can be quite involved as this application note details.

Several stack placement strategies are commonly used. One strategy is to place the stack at the top of RAM and let it grow downward without any bound. That strategy does nothing to guard against a potential deeply recursive function working all the way down RAM and overwriting program data. Another strategy is to define the bottom limit of the stack to be the beginning of RAM and place the initial top of the stack somewhere higher up, say 4KiB, in memory. Since memory below the beginning of RAM is usually not present, this can cause a fault if the stack grows too large. This strategy is adopted here. The process stack is placed at the bottom of RAM and the main stack located immediately next to it, higher in memory.

It is difficult to estimate up front how large the stack should be. That's one of the reasons to fill the stack space with an unusual number that will allow you to get an estimate of usage. Start with 4KiB and make provisions to change the size with a pre-processor symbol that can be defined as an option to the compiler.

```
<<startup_apollo3: constants>>=
#ifndef DEFAULT_STACK_SIZE
#   define DEFAULT_STACK_SIZE   0x1000
#endif /* DEFAULT_STACK_SIZE */

#ifndef MAIN_STACK_SIZE
#   define MAIN_STACK_SIZE   DEFAULT_STACK_SIZE
#endif /* MAIN_STACK_SIZE */

#ifndef PROCESS_STACK_SIZE
#   define PROCESS_STACK_SIZE   DEFAULT_STACK_SIZE
#endif /* PROCESS_STACK_SIZE */
```

The stack is then just an array variable of the desired size and the linker places the memory where it is directed.

```
<<startup_apollo3: static data>>=
static uint8_t alignas(uint64_t) main_stack[MAIN_STACK_SIZE]
        __attribute__((used,section(".main_stack"))) ;          // ❶
```

❶ Another information packed definition.

1. The stack is aligned on an 8 byte boundary to comply with AAPCS requirements.

2. Since the stack variable name is not otherwise referenced in this file, the attribute forces the compiler to keep the variable anyway, especially since it is marked `static`. The compiler is otherwise within its rights to discard file static variables which are never accessed in the code of the defining file.
3. Place the `main_stack` in the `.main_stack` segment so you can tell the linker where to locate the stack in memory.

```
<<startup_apollo3: static data>>=
static uint8_t alignas(uint64_t) process_stack[PROCESS_STACK_SIZE]
        __attribute__((used,section(".process_stack"))) ;
```

To use the convenience macros for alignment, the standard header file is required.

```
<<startup_apollo3: include files>>=
#include <stdalign.h>
```

Initializing stack memory is similar to initializing the BSS section. However, there are two other consideration that must be attended to:

1. The startup code itself is executing using the Main Stack. You must not overwrite any of the Main Stack that is in use. This means you must *not* write past the address contained in the MSP.
2. You want to write the bottom of the stack with a value which is different than the value used for filling the remaining stack memory. The different value can be used, potentially, to calculate the stack usage through a debugging interface.

```
<<init_ram: stack sections>>=
InitSegmentDesc const *stack_seg = &__stack_table_start__ ;
InitSegmentDesc const *const stack_seg_end = &__stack_table_end__ ;

uint32_t *const msp = (uint32_t *const)__get_MSP() ;                // ❶

for ( ; stack_seg < stack_seg_end ; stack_seg++) {
    uint32_t *dst = stack_seg->dst ;
    uint32_t *const segment_end = &dst[stack_seg->nwords] ;
    uint32_t *const dst_end = (msp > dst && msp < segment_end) ?
            msp : segment_end ;                                     // ❷

    if (dst < dst_end) {
        *dst++ = UINT32_C(0xdeadbeef) ;                             // ❸
    }

    while (dst < dst_end) {
        *dst++ = UINT32_C(0x78563412) ;
    }
}
```

❶ Since the stack pointer is in play, *i.e.* it is currently being used, you must make sure not to overwrite the portion of the stack that is in use.

❷ The ternary operator makes it possible to declare the pointer value as `const`.

❸ There are a few “cute” words which can be spelled with the alphabetic characters used for hexadecimal numbers. All that is required is something distinctive to recognize the boundary and it is useful if each byte of the 4 byte word is a different value.

Memory Which Survives Reset

Just as there are `.data` and `.bss` sections, memory intended to survive reset also needs to be split between sections which are explicitly initialized and zero initialized. So, you define the `.noinit_data` and the `.noinit_bss` sections.

The memory sections that are intended to survive reset must have their initialization conditioned upon whether an actual Power On Reset (POR) occurred. If a POR occurred, then the memory locations are set to zero. Otherwise, the memory is left in tact.

For the Apollo 3 chip, the reset generator peripheral has a register which holds indicators of the source and type of the reset. This is the information used to determine if there was a POR.

There are two other considerations:

- A *magic* number is written into the first word of the memory section to indicate that the initialization is complete. That magic value is expected always to be there and if it is not, then the memory is reinitialized regardless of the POR status.
- The Apollo 3 data sheet indicates that the reset generator status register value is *not* retained when deep sleep is entered. To make it available to the application, it is stored in the *.noinit_bss* section.

These considerations are captured in a variable. The variable serves to give the section a size and other parts of the system may declare variables in the `.noinit_data` or `.noinit_bss` sections to store things they wish to survive reset. One good use would be to store the conditions detected during a major system exception such as a Hard Fault.

```
<<linker symbols: data type declarations>>=
typedef struct {
    uint32_t magic ;
    uint32_t reset_status ;
} NoinitSegmentStatus ;
```

The code to initialize the segments combines the tables created by the linker (as was used in the other segments) along with the `noinit_status` variable.

```
<<init_ram: no init sections>>=
uint32_t const magic = 0xe4fda777U ;                                    // ❶

HDWR_Register reset_status = RSTGEN->STAT ;
bool por = BTWD_FIELD_TEST(reset_status, RSTGEN_STAT_PORSTAT) ||
        BTWD_FIELD_TEST(reset_status, RSTGEN_STAT_POIRSTAT) ;           // ❷

if (por || noinit_status.magic != magic) {
    CopySegmentDesc const *noinit_copy_seg = &__noinit_data_copy_table_start__ ;
    CopySegmentDesc const *const noinit_copy_seg_end =
            &__noinit_data_copy_table_end__ ;

    for ( ; noinit_copy_seg < noinit_copy_seg_end ; noinit_copy_seg++) {
        uint32_t const *src = noinit_copy_seg->src ;
        uint32_t const *const src_end = &src[noinit_copy_seg->nwords] ;
        uint32_t *dst = noinit_copy_seg->dst ;

        while (src < src_end) {
            *dst++ = *src++ ;
        }
    }

    InitSegmentDesc const *noinit_bss_seg = &__noinit_bss_table_start__ ;
    InitSegmentDesc const *const noinit_bss_seg_end = &__noinit_bss_table_end__ ;

    for ( ; noinit_bss_seg < noinit_bss_seg_end ; noinit_bss_seg++) {
        uint32_t *dst = noinit_bss_seg->dst ;
        uint32_t *const dst_end = &dst[noinit_bss_seg->nwords] ;

        while (dst < dst_end) {
```

```
            *dst++ = UINT32_C(0) ;
        }
    }

    noinit_status.magic = magic ;
}

noinit_status.reset_status = reset_status ;
```

❶ An arbitrary, improbable number that is used as the initialization indicator.

❷ For the Apollo 3 SOC, the reset generator peripheral allows for software to generate the same power on initialization signal as would be generated in hardware. The reset status keeps separate status for the two conditions, but you want to have the same behavior for both.

Privileged Execution

Before transferring control to `main()`, two bits in the `CONTROL` register were changed that have significant design implications.

- Change the stack pointer for Thread mode operation to use the PSP.
- Set Thread mode operation to be unprivileged.

At reset, the processor executes in Thread mode (*cf.* Handler mode) using the MSP and privileged execution. This is the simplest mode of operation and allows unfettered access to the system by any code. Many systems are run in this configuration.

In this design, both stacks are used. The reasoning for a two stack approach was discussed previously.

A more complex set of trade-offs is associated with executing the application code in unprivileged mode. By placing restrictions on the application code, the intent is to build a more constrained execution environment. Small software errors can easily send a microcontroller executing in an uncontrolled manner. The intent for a more controlled execution environment is to be able to catch unwanted behavior earlier in the development process. When a microcontroller system is deployed, there is usually little recovery opportunity should things go terribly wrong. You would like to be able to catch such behavior early and if it does happen in a deployed system to limit the potential for damage. By excluding the application code from operating with privilege, the problem is divided in half by insuring that any computation on critical system registers or peripherals cannot happen in the application by means of a stray pointer.

The topic of constraining execution of the software is a recurrent theme in this book.

One trade-off for erecting an execution barrier between the system and the application is that it is computationally more costly to control the peripheral devices. Requiring privilege and protecting the memory mapped I/O regions means that all access to system and peripheral registers must happen via the exception handling mechanism. That mechanism involves at least pushing and popping an exception stack frame for any system level access. Effectively, building a wall between the system and application processing implies scaling that wall whenever the two sides need to interact. It is simply the price that must be paid for a more robust system. Whether the increased cost of privileged access is a performance problem is determined both by the exception overhead and the frequency at which the overhead is incurred. This can only truly be determined by measurement. Good design practice attempts to minimize application code doing intensive system interactions in performance critical areas. There are usually effective ways to design the interactions between the unprivileged and privileged code so as not to impose an undue burden.

Note that the intent here is not to impose some level of security on the system to prevent malicious intent. Microcontroller systems typically execute a single program and, if field updates are protected from malicious intervention, our intent is to mitigate the effects of software bugs. Bugs are to software as noise is to electronics — something that does not go completely away despite your best efforts at control.

Startup Code Layout

The code for start up is placed in the `startup_apollo3.c` file. This section shows the order of the literate program chunks that are put together to create the source file that is given to the compiler.

```
<<startup_apollo3.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Ambiq Apollo 3 startup code
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
<<startup_apollo3: include files>>
/*
 * Constants
 */
<<startup_apollo3: constants>>
/*
 * Data Type Declarations
 */
<<startup_apollo3: data type declarations>>
/*
 * External Declarations
 */
<<startup_apollo3: external declarations>>
/*
 * Forward References
 */
<<startup_apollo3: forward references>>
/*
 * Static Data
 */
<<startup_apollo3: static data>>
/*
 * Static Functions
 */
<<startup_apollo3: static functions>>
/*
 * External Data
 */
<<startup_apollo3: exception table definition>>
<<startup_apollo3: noinit data definition>>
<<startup_apollo3: main argument definitions>>
/*
 * External Functions
 */
<<startup_apollo3: external functions>>
```

Start Main code file

The code that invokes the `main()` function is in a separate file as mentioned previously.

```
<<start_main.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Starting the main function
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
#include <stdlib.h>
#include <stdnoreturn.h>
#include "apollo3.h"
#include "linker_symbols.h"
#include "bit_twiddle.h"
/*
 * External Declarations
 */
<<start main: main argument definitions>>
extern void main(int argc, char **argv) ;
/*
 * External Functions
 */
<<start main: external functions>>
```

Core Initialization

In the previous section, the focus was on memory, both laying out the exception vector table and initializing RAM memory to suit our needs. In this section, the focus shifts to getting the processor into the state needed to execute any program. The intent is to initialize only those aspects of the hardware which are essential to program execution. Other hardware initialization is deferred until after `main()` is reached. This code is intended to be generally applicable to the class of applications expected to be deployed and at start up exactly what other hardware might be used by a program is not known.

CMSIS conventions are to supply two files that handle SOC specific code sequences.

1. `startup_<device>.c` is meant to do just what it says, start up the computer. This is the file created in the previous chapter.
2. `system_<device>.c` is a file which contains implementations of functions that specific to the SOC. At a minimum, CMSIS specifies that the functions `SystemInit()` and `SystemCoreClockUpdate()` be supplied. For most vendor supplied `system_<device>.c` files, that's all you get.

`SystemInit()`

The `SystemInit()` function is intended to be invoked first in the reset handler to put the computer is its intended operational state.

For example, most microcontrollers have multiple ways to supply clocks to the core and peripherals. Often, an external high speed crystal oscillator is used to clock the processor. But such crystals need to be started up and there is a period of time where

they must stabilize. To get the processor running at all, there is usually an internal RC oscillator which runs at a lower frequency, but is ready for use immediately. These are the types of functions allocated to `SystemInit()`.

There is one other detail about `SystemInit()` which is easily overlooked. At the time `SystemInit` is invoked, memory has not been set up according to the expectation of a standard "C" program. This makes some sense as you are just trying to get things to run in a stable configuration. After `SystemInit()` returns and before `main()` is invoked, the initialized data section and the zero-initialized data section are set up as shown in the previous section. The consequence is that assigning values to global variables has no effect since the read/write data areas of the program will be modified after `SystemInit` returns.

As with the startup code, to access Core CMSIS definitions, the SOC header file is needed.

```
<<system_apollo3: include files>>=
#include "apollo3.h"
#include "linker_symbols.h"
```

The following are the areas of general interest for the class of programs intended to run:

1. Clock frequency of the processor core.
2. Flash memory cache.
3. Vector table offset.
4. Processor controls.
5. Floating point unit (FPU).
6. Exception priorities.
7. Memory protection unit (MPU).

```
<<system_apollo3: external function declarations>>=
extern void SystemInit(void) ;
```

The `SystemInit` function initializes processor specific controls to ready the system for execution. This is a standard CMSIS function and is called immediately after entry into the system Reset exception handler.

SystemInit Implementation

```
<<system_apollo3: external functions>>=
__WEAK void
SystemInit(void)
{
    //
    // Initialize the system
    // Do not use global variables because this function is called before
    // reaching pre-main. Read/write sections maybe overwritten afterwards.
    //

    <<SystemInit: high frequency clock>>
    <<SystemInit: enable flash cache>>
    <<SystemInit: vector table>>
    <<SystemInit: system timer>>
    init_ram() ;
    SystemControlsInit() ;
    SystemFPUInit() ;
    SystemExcPriorityInit() ;
    SystemMemProtectInit() ;
}
```

Note the function is marked *weak* so it may be overridden. However, this implementation covers the major areas. The other initialization functions invoked by `SystemInit()` are also designated as *weak*, so it is possible to override only part of the initialization sequence.

For the rest of this section, the discussion is specific to the Apollo 3 Blue SOC. Some familiarity with the data sheet is essential for a complete understanding, but the general functions dealt with here are present in other microcontrollers.

High Frequency Clock

According to the data sheet, the reset value of the core clock runs the High Frequency clock at half speed. Here it is adjusted up to full speed. The effects on power consumption are difficult to predict. The higher speed clock consumes more power, but the processor can finish computations more quickly and return to lower power modes. Measurement on an actual running application are necessary if ultra-low power operation is needed.

```
<<SystemInit: high frequency clock>>=
CLKGEN->CLKKEY = CLKGEN_CLKKEY_CLKKEY_Key ; // ❶
CLKGEN->CCTRL = CLKGEN_CCTRL_CORESEL_HFRC ; // Div by 1 for 48MHz
CLKGEN->CLKKEY = 0 ;
```

❶ It is necessary to unlock (and later relock) the clock generator registers to change them. This protects against accidental changes.

Flash Memory Cache

To speed up the access to instructions and data obtained from flash memory, the Apollo 3 has a cache which can return data without any wait states on a cache hit. The setup code here is cribbed from the Ambiq SDK function, `am_hal_cachectrl_config()` using values from the default configuration as well as recommended values from the data sheet. The cache controller is a sophisticated peripheral and only a subset of its capabilities are used here.

```
<<SystemInit: enable flash cache>>=
uint32_t cache_cfg = 0 ;

cache_cfg = BTWD_FIELD_SET(cache_cfg, CACHECTRL_CACHECFG_DATA_CLKGATE) ;
cache_cfg = BTWD_FIELD_SET(cache_cfg, CACHECTRL_CACHECFG_CACHE_CLKGATE) ;
cache_cfg = BTWD_FIELD_SET(cache_cfg, CACHECTRL_CACHECFG_DCACHE_ENABLE) ;
cache_cfg = BTWD_FIELD_SET(cache_cfg, CACHECTRL_CACHECFG_ICACHE_ENABLE) ;
cache_cfg = BTWD_FIELD_INSERT(cache_cfg, CACHECTRL_CACHECFG_CONFIG_W1_128B_1024E,
        CACHECTRL_CACHECFG_CONFIG) ;

CACHECTRL->CACHECFG = cache_cfg ;

uint32_t flash_cfg = 0 ;

flash_cfg = BTWD_FIELD_INSERT(flash_cfg, CACHECTRL_FLASHCFG_LPMMODE_STANDBY,
        CACHECTRL_FLASHCFG_LPMMODE) ;
flash_cfg = BTWD_FIELD_INSERT(flash_cfg, CACHECTRL_FLASHCFG_SEDELAY_RECOMMENDED,
        CACHECTRL_FLASHCFG_SEDELAY) ;
flash_cfg = BTWD_FIELD_INSERT(flash_cfg, CACHECTRL_FLASHCFG_RD_WAIT_RECOMMENDED,
        CACHECTRL_FLASHCFG_RD_WAIT) ;

CACHECTRL->FLASHCFG = flash_cfg ;

BTWD_SET_REG_FIELD(&CACHECTRL->CACHECFG, CACHECTRL_CACHECFG_ENABLE) ;

BTWD_SET_REG_FIELD(&PWRCTRL->MEMPWDINSLEEP, PWRCTRL_MEMPWDINSLEEP_CACHEPWDSLP) ;    // ❶
```

❶ An Ambiq report (AMA3B1KK-xxx) suggests powering down the cache in deep sleep mode to avoid any potential of corruption of data in the cache during deep sleep.

Vector Table Offset

To insure that the processor uses the IRQ handlers defined here, the `VTOR` register is set to point to the exception vector table.

```
<<SystemInit: vector table>>=
extern uint32_t __Vectors[] ;
SCB->VTOR = (uint32_t)__Vectors ;
__DSB() ;                                          // ❶
```

❶ This is the Data Synchronization Barrier instruction. It insures that all the data transfers, to the SCB block in this case, are complete before the next instruction is executed. For the Cortex-M4, the SCB is *strongly ordered* and so the `DSB` is not strictly necessary. It is included based on ARM architectural recommendations. The best guide to these situations is the *ARM Cortex-M Programming Guide to Memory Barrier Instructions: Application Note 321*.

System Timer

The Apollo 3 has a system timer peripheral which is used for timestamping and other purposes. Note this is different from the `SysTick` core peripheral. `SysTick` is not used in this design. Its clock rate is too high and it is only 24 bits long. The intended use for `SysTick` is as timer for round-robin task scheduling. This design does not use any tasks so scheduling is unnecessary.

The system timer peripheral is initialized here so that it is available as a source of timestamps, particularly for fault handlers.

```
<<SystemInit: system timer>>=
uint32_t config = CTIMER->STCFG ;
bool timer_frozen = BTWD_FIELD_TEST(config, CTIMER_STCFG_FREEZE) ;
uint32_t clk = BTWD_FIELD_EXTRACT(config, CTIMER_STCFG_CLKSEL) ;
if (timer_frozen || clk == CTIMER_STCFG_CLKSEL_NOCLK) {                    // ❶
    BTWD_SET_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_FREEZE) ;
    BTWD_WRITE_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_CLKSEL_XTAL_DIV1,
            CTIMER_STCFG_CLKSEL) ;
    BTWD_SET_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_CLEAR) ;               // ❷
    BTWD_CLEAR_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_CLEAR) ;
    BTWD_SET_REG_FIELD(&CTIMER->STMINTEN, CTIMER_STMINTEN_OVERFLOW) ;      // ❸
    BTWD_CLEAR_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_FREEZE) ;
}
NVIC_EnableIRQ(STIMER_IRQn) ;                                              // ❹
```

❶ At power on reset, the timer is “frozen.” At other types of reset, its state survives and further initialization is not required.

❷ The “clear” bit field must be set to one then cleared to zero to cause the counter value itself to be zeroed.

❸ The timer overflow interrupt is used to track larger time counts in special purpose registers which also survive reset.

❹ Any type of reset disables the overflow interrupt at the `NVIC`.

The timer overflow is handled by adding to the SNVR[0-3] registers to keep the extra time precision in system timer registers.

```
<<system_apollo3: external functions>>=
void
STIMER_IRQHandler(void)
{
    if (BTWD_TEST_REG_FIELD(&CTIMER->STMINTSTAT, CTIMER_STMINTSTAT_OVERFLOW)) {
        CTIMER->STMINTCLR = CTIMER_STMINTCLR_OVERFLOW_Msk ;
        CTIMER->SNVR[0] += 1 ;
        if (CTIMER->SNVR[0] == 0) {
            CTIMER->SNVR[1] += 1 ;
            if (CTIMER->SNVR[1] == 0) {
                CTIMER->SNVR[2] += 1 ;
                if (CTIMER->SNVR[2] == 0) {
```

```
                    CTIMER->SNVR[3] += 1 ;
                }
            }
        }
    } else {
        NVIC_DisableIRQ(STIMER_IRQn) ;              // ❶
    }
}
```

❶ The overflow and capture registers share the main STIMER_IRQn interrupt vector. Currently, this design does not use the capture registers. If some interrupt other than an overflow arrives, then somehow the capture register interrupts were enabled without modifying this IRQ handler. The fallback is just to shut the whole thing down. The compare registers of the system timer peripheral have their own interrupt vectors.

`SystemCoreClockUpdate()`

```
<<system_apollo3: external function declarations>>=
extern void SystemCoreClockUpdate(void) ;
```

```
<<system_apollo3: external data declarations>>=
extern uint32_t SystemCoreClock ;
```

The `SystemCoreClockUpdate` function updates the value of the global variable, `SystemCoreClock` to be the frequency of the clock running the processor in units of Hertz. This is a standard CMSIS function. System code should invoke this function after any change to the frequency of the core clock. Applications may use the global value, `SystemCoreClock`, to compute frequency information for other clocks in the system which may be derived from the core clock frequency.

SystemCoreClockUpdate Implementation

```
<<system_apollo3: external functions>>=
void
SystemCoreClockUpdate(void)
{
    //
    // Calculate the system frequency based upon the current register settings.
    // This function can be used to retrieve the system core clock frequency
    // after user changed register sittings.
    //
    SystemCoreClock = __SYS_OSC_CLK /
            (BTWD_READ_REG_FIELD(&CLKGEN->CCTRL, CLKGEN_CCTRL_CORESEL) + 1) ;
}
```

```
<<system_apollo3: external data>>=
uint32_t SystemCoreClock = __SYSTEM_CLOCK;  // System Clock Frequency (Core Clock) ❶
```

❶ Note that when the initialized data is copied from flash memory to RAM during RAM initialization, the core clock frequency value will match the effect of `SystemCoreClockUpdate()` function.

Some constant values which match the clock frequencies of the Apollo 3 are required.

```
<<system_apollo3: constants>>=
#define __HSI             (6000000UL)       // ❶
#define __XTAL            (32768UL)         // Crystal Oscillator frequency
#define __SYS_OSC_CLK     (48000000)        // Main oscillator frequency
#define __SYSTEM_CLOCK    (1*__SYS_OSC_CLK)
```

❶ These constants come directly from the Apollo 3 SDK.

SystemControlsInit()

SystemControlsInit

```
<<system_apollo3: external function declarations>>=
extern void SystemControlsInit(void) ;
```

The `SystemControlsInit()` function initializes system registers which control the actions taken by the processor in response to situations created by the running program.

SystemControlsInit Implementation

```
<<system_apollo3: external functions>>=
__WEAK
void
SystemControlsInit(void)
{
    stwd_sp_align_8byte(true) ;             // ❶
    stwd_enable_div_zero_trap(true) ;
    stwd_enable_unaligned_trap(true) ;
    stwd_enable_deep_sleep(true) ;
}
```

❶ Enabled by default on a Cortex-M4.

The Cortex-M4 core has a number of controls that affect aspects of behavior is enforced for all application programs.

- Stack alignment is set to 8 bytes to conform to AAPCS requirements and conventional usage. Care must be taken that stack areas are 8 byte aligned.
- Division by zero is not allowed. Any code which divides must either know by context that the divisor is non-zero or must make a test of zero to prevent the condition.
- Unaligned memory accesses are not allowed. *N.B.* that the compiler flag `-mno-unaligned-access` must be given to the compiler to prevent it from generating code which has unaligned accesses. Unaligned access can result from *up casting* a pointer from a lesser alignment to a greater alignment. This is an indication of a program design issue.
- Enabling deep sleep mode in the processor requires a control bit to be set.

The prototypes for the `stwd_xxx` functions are contained in the corresponding header file.

```
<<system_apollo3: include files>>=
#include "sys_twiddle.h"
```

SystemFPUInit()

```
<<system_apollo3: external function declarations>>=
extern void SystemFPUInit(void) ;
```

The `SystemFPUInit()` function initializes the system registers associated with the FPU to configure access and stack context behavior.

SystemFPUInit Implementation

```
<<system_apollo3: external functions>>=
__WEAK
void
SystemFPUInit(void)
{
    stwd_set_fpu_access(fpu_access_full) ;              // ❶
    stwd_set_fpu_stack_context(fpu_no_auto_stack) ;     // ❷
}
```

❶ By default after reset, floating point operations are not enabled. Here, they are enabled. The `stwd_set_fpu_access()` function contains logic to decide if the floating point option is present and if the compiler flags indicate that it is used.

❷ This setting is appropriate when there is no task context switching and none of the IRQ handlers use floating point. This describes the types of the targeted applications.

The Cortex-M4 has a single precision floating point unit. It is intended for use by application code and in this particular use case, no floating point operations are allowed from IRQ handlers.

`SystemExcPriorityInit()`

```
<<system_apollo3: external function declarations>>=
extern void SystemExcPriorityInit(void) ;
```

The `SystemExcPriorityInit()` function initializes the priorities of all configurable priority exceptions. Additionally, this function enables system faults.

SystemExcPriorityInit Implementation

```
<<system_apollo3: external functions>>=
__WEAK
void
SystemExcPriorityInit(void)
{
    stwd_enable_bus_fault(true) ;
    stwd_enable_usage_fault(true) ;
    dtwd_enable_fault_capture(true) ;           // ❶
}
```

❶ When enabled, the Apollo 3 has some additional fault registers that it can record.

At a minimum, the default implementation enables the Bus Fault and Usage Fault. Should anything happen to cause one of these faults and it is *not* enabled, the fault is escalated to become a Hard Fault. This function will be replaced when the execution priority scheme is presented.

There are a set of device specific functions for peripheral register manipulation and the header file must be included.

```
<<system_apollo3: include files>>=
#include "dev_twiddle.h"
```

The `dev_twiddle` functions are discussed in a following section.

SystemMemProtectInit()

```
<<system_apollo3: external function declarations>>=
extern void SystemMemProtectInit(void) ;
```

The `SystemMemProtectInit()` function initializes the Memory Protection Unit. Once initialized is also enables the Memory Management fault.

MPU initialization is deferred to a later chapter. The subject of memory protection requires considerable discussion which would only be a further distraction just now.

SystemMemProtectInit Implementation

```
<<system_apollo3: external functions>>=
__WEAK
void
SystemMemProtectInit(void)
{
}
```

SystemMissingHandler()

The `SystemMissingHandler()` function is called by the Default exception handler to provide an opportunity to diagnose and/or record the occurrence of an exception for which no handler was provided.

```
<<system_apollo3: external function declarations>>=
extern noreturn void
SystemMissingHandler(
    ExceptionFrame *exc_frame,
    uint32_t exc_return,
    uint32_t ipsr) ;
```

exc_frame	A pointer to the exception stack frame generated when the processor enters exception execution.
exc_return	The exception return value for the exception. The encoding of the exception return values indicates the conditions which are re-established when the exception returns.
ipsr	The value of the IPSR status register giving the exception number of the currently executing exception.

The default implementation is to invoke the abnormal ending function.

SystemMissingHandler Implementation

```
<<system_apollo3: external functions>>=
__attribute__((used,weak))                              // ❶
noreturn void
SystemMissingHandler(
    ExceptionFrame *exc_frame,
    uint32_t exc_return,
    uint32_t ipsr)
{
    SystemAbend() ;                                      // ❷
}
```

❶ Tell the compiler this function is actually "`used`". Since it only appears in inline assembly, GCC does not know that it was actually used by any code. Also, this function is given *weak* linkage so it may be overridden.

❷ **N.B.** no recovery is attempted here. The default implementation simply abnormally terminates the program.

System Code layout

This section shows how the content of the files `system_apollo3.c` and `system_apollo3.h` is composed from literate program chunks.

System code file

```
<<system_apollo3.c>>=
<<edit warning>>
<<copyright info>>
<<ambiq copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Ambiq Apollo 3 CMSIS system code
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
<<system_apollo3: include files>>
/*
 * Constants
 */
<<system_apollo3: constants>>
/*
 * Forward References
 */
<<system_apollo3: forward references>>
/*
 * External Data
 */
<<system_apollo3: external data>>
/*
 * Static Functions
 */
<<system_apollo3: static functions>>
/*
 * External Functions
 */
<<system_apollo3: external functions>>
```

System include file

```
<<system_apollo3.h>>=
<<edit warning>>
<<copyright info>>
<<ambiq copyright info>>
/*
 *++
```

```
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Ambiq Apollo 3 CMSIS system code header file
 *--
 */
#ifndef SYSTEM_APOLLO3_H_
#define SYSTEM_APOLLO3_H_

/*
 * Include files
 */
#include <stdint.h>
#include <stdnoreturn.h>
/*
 * Data Type Declarations
 */
<<system_apollo3: data type declarations>>
/*
 * External Data
 */
<<system_apollo3: external data declarations>>
/*
 * External Functions
 */
<<system_apollo3: external function declarations>>

#endif /* SYSTEM_APOLLO3_H_ */
```

Scripting the Linker

The last steps to getting a program to run involve scripting the linker. Let's recap what has been done have so far.

- The exception vector table provides the Cortex-M4 core the necessary data to find the Reset handler function. The table was defined to include all the exception handlers and made arrangements for it to be populated either by a default function or one found elsewhere in the program code. The Reset handler function itself is provided and cannot be overridden.
- The Reset handler first initializes those aspects of the processor core required to execute any program.
- After the core is executing as intended, the RAM memory is initialized as required by the "C" language standard and to suit the needs of the applications.
- With the core initialized and RAM set up properly, `main()` is called to start the application.

One nagging detail remains. Where in memory is the exception vector table and all the other code located? It's essential to be able to compile pieces separately. But with that flexibility comes the need to put the pieces back together, and that is what a linker does.

Linker sections have an additional complication. It has to have a name, and other pieces of code will ultimately need to refer to that name. The usual solution is to come up with a naming convention. "C" compilers use linker sections to keep code, initialized data, and uninitialized data separate. The names used are conventions imposed by the compiler. For other specialized sections, naming is arbitrary, but long standing conventions should be respected. Following established convention, the name, `.vectors`, has been given to the to exception vector table.

Linker defined symbols must also be handled. For convenience, external declarations for the linker symbols have been placed in a header file. Using literate program chunks, the linker script, which defines the symbol values, is kept lexically close to the "C" declarations, which allows the start up code to use those symbols. This makes the correspondence between them readily apparent.

Linker script layout

A linker script must define:

a. where to begin execution,

b. the addresses of the different types of memory, and

c. how that memory is filled with code and data.

```
<<apollo3.ld>>=
<<copyright info>>
<<linker script: entry point>>
<<linker script: memory layout>>
<<linker script: section descriptions>>
```

Entry point

In this environment, the program starts at the Reset Handler.

```
<<linker script: entry point>>=
ENTRY(Reset_Handler)
```

Memory layout

The following table shows the memory map for the Apollo 3 SOC. Note that RAM starts at 0x1000_0000 and that the first 64 KiB are Tightly Coupled Memory (TCM).

Table 2.1: Apollo 3 System Memory Map

Address	Name	Executable	Description
0x00000000 — 0x000FFFFF	Flash	Y	Flash Memory
0x00100000 — 0x03FFFFFF	Reserved	X	No device at this address range
0x04000000 — 0x07FFFFFF	External MSPI Flash	Y	XIP Read-Only External MSPI Flash
0x08000000 — 0x08000FFF	Boot Loader ROM	Y	Execute Only Boot Loader and Flash Helper Functions.
0x08001000 — 0x0FFFFFFF	Reserved	X	No device at this address range
0x10000000 — 0x1000FFFF	SRAM (TCM)	Y	Low-power / Low Latency SRAM (TCM)
0x10010000 — 0x1005FFFF	SRAM (Main)	Y	Main SRAM
0x10060000 — 0x3FFFFFFF	Reserved	X	No device at this address range
0x40000000 — 0x50FFFFFF	Peripheral	N	Peripheral devices
0x51000000 — 0x51FFFFFF	External Memory	X	Read/Write External Memory (MSPI) [Chip Rev B Only]
0x52000000 — 0xDFFFFFFF	Reserved	X	No device at this address range
0xE0000000 — 0xE00FFFFF	PPB	N	NVIC, System timers, System Control Block

Table 2.1: (continued)

Address	Name	Executable	Description
0xE0100000 — 0xEFFFFFFF	Reserved	X	No device at this address range
0xF0000000 — 0xF0000FFF	Debug ROM	N	Debug ROM
0xF0001000 — 0xFFFFFFFF	Reserved	X	No device at this address range

The SparkFun board being used does not have any external memory. So, flash and RAM are the only two interesting areas of the memory map.

```
<<linker script: memory layout>>=
MEMORY
{
  FLASH (rx) : ORIGIN = 0x00010000, LENGTH = 960K
  RAM (rwx) : ORIGIN = 0x10000000, LENGTH = 384K
}
```

Note
SparkFun has installed a secondary bootloader into the part. To preserve the secondary bootloader, the flash origin is set to 0x0001_0000. To overwrite the SparkFun bootloader, set the RAM origin at 0x0000_C000.

Linker sections

This section shows the linker commands which place the objects into memory. The order here is flash first and RAM second. The RAM layout order follows that of the previous discussion about the code which performs the RAM initialization.

The following diagram shows the layout in graphical form.

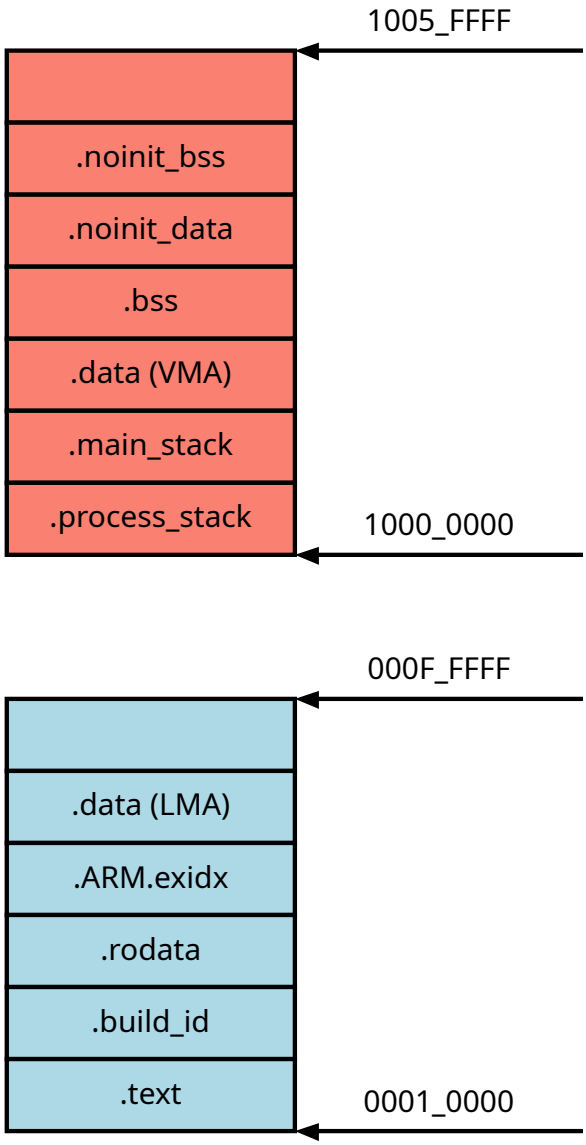

Figure 2.3: Linker Section Memory Layout

The linker script to layout the code contains a set of output section descriptions in the same order.

```
<<linker script: section descriptions>>=
SECTIONS
{
    <<linker section: text>>
    <<linker section: build_id>>
    <<linker section: rodata>>
    <<linker section: ARM.exidx>>
    <<linker section: process stack>>
    <<linker section: main stack>>
    <<linker section: data>>
    <<linker section: bss>>
    <<linker section: noinit data>>
    <<linker section: noinit bss>>

    PROVIDE(end = .) ;          /* ❶ */
}
```

❶ The `nosys` library from `newlib` library provides stubs for the system related functions which `newlib` assumes. One of those system functions is `sbrk()`, which is used by `malloc()` and related functions to acquire memory for a heap. The `sbrk()` function assumes that the linker emits a symbol which marks the end of memory allocated by the program. The `sbrk()` function uses memory for `end` upward to provide the memory for a heap. Although there are no calls to `malloc` in this code, there are internal calls by `newlib`. By providing this symbol, `newlib` can be linked using `nosys` to obtain all the standard library functions.

Text section

The `.text` section includes all the object code for the program. The exception vector table is placed first so it is located at the origin of the flash memory.

```
<<linker section: text>>=
.text : ALIGN(4)
{
    __text_start__ = . ;

    KEEP(*(.vectors*))
    KEEP(*(.ble_patch*))

    . = ALIGN(4) ;
    *(.text*)

    . = ALIGN(4) ;
    __text_end__ = . ;
} > FLASH
```

The corresponding external declarations for the linker symbols is then given by:

```
<<linker symbols: external declarations>>=
extern uint32_t const __text_start__ ;
extern uint32_t const __text_end__ ;
```

Build id section

The `.build_id` section loads the contents of a *note* section which contains an identifier of the executable. To obtain a build id in a section requires an explicit request to the linker by using the `--build-id` linker command option.

```
<<linker section: build_id>>=
.build_id : ALIGN(4)
{
    PROVIDE(SystemBuildIDNote = .) ;
    *(.note.gnu.build-id)
} > FLASH
```

```
<<linker symbols: external declarations>>=
extern const ElfNoteSectionDesc SystemBuildIDNote ;
```

The linker places the build ID in a *note* section. The layout of data in that section contains several elements. Again, a “C” structure declaration must be constructed that matches the layout data as the linker generates it. The following structure declaration shows the components.

```
<<linker symbols: data type declarations>>=
typedef struct {
    uint32_t namesz ;
    uint32_t descsz ;
    uint32_t type ;
    uint8_t data[] ;
} ElfNoteSectionDesc ;
```

The linker packs naming and descriptive information together into the variable sized `data` member. The offset in `data` to the build ID is given by the `namesz` member.

Read only data section

The compiler places read-only data in a separate section from executable code. Ultimately, the MPU is used to prevent executing the read-only data. This just makes that step a bit easier when it is introduced later.

In addition to the read only data sections emitted by the compiler, the initialization copy tables are placed in the same section. These tables are read only data, but are generated by the linker rather than the compiler.

```
<<linker section: rodata>>=
.rodata : ALIGN(4)
{
    __rodata_start__ = . ;

    *(.rodata*)

    . = ALIGN(4) ;

    <<linker section: memory init tables>>

    . = ALIGN(4) ;
    __rodata_end__ = . ;
} > FLASH
```

```
<<linker symbols: external declarations>>=
extern uint32_t const __rodata_start__ ;
extern uint32_t const __rodata_end__ ;
```

Memory Initialization Tables

In a previous section, linker defined tables were used to describe the memory regions which needed initialization. Here, the linker commands required to build the copy table values are described. There are five areas of RAM which require specialized initialization.

```
<<linker section: memory init tables>>=
<<linker tables: data copy>>
<<linker tables: bss zero>>
<<linker tables: stack>>
<<linker tables: noinit data copy>>
<<linker tables: noinit bss zero>>
```

To copy a section of memory to another location, the address of the source, the address of the destination, and the amount of data to copy are needed. The linker is requested to place the required information into the linked output. To be useful to the code, the sequence of 32-bit values placed into memory by the linker must match the data type declaration that is used by the code. *N.B.* assumptions about the alignment and padding of structure members used by the compiler are being made. These are safe assumptions in this case since the Cortex-M4 has a regular 32-bit word size that does not need any padding. Still, the linker and compiler do not know about this correspondence and the responsibility to insure it is correct falls upon us.

Data copy table values

```
<<linker tables: data copy>>=
 __data_copy_table_start__ = . ;

LONG(LOADADDR(.data))                /* ❶ */
LONG(ADDR(.data))                    /* ❷ */
LONG(SIZEOF(.data) / 4)

 __data_copy_table_end__ = . ;
```

❶ The `LOADADDR()` operator returns the address where the section is loaded in memory.

❷ The `ADDR()` operator returns the address where the section resides during execution.

In the previous linker script segment, the `LONG` command places a 32-bit value into the output section. These commands also illustrate the linker concepts of *virtual memory address* (VMA)[6] and *load memory address* (LMA). The VMA is the address where the data is assumed to reside when the program is executing. The LMA is the address where the section data is placed before execution begins, *i.e.* where it is loaded in memory by the executable file. For all other sections, these two addresses are the same. But for the `.data` section, the initializer values must be copied from flash, *i.e.* from the LMA, to where they are referenced by the program code, *i.e.* to the VMA.

When the linker resolves symbols between code and data, the resolution happens in VMA terms. Setting the LMA to be different from the VMA and then placing the LMA of the `.data` section into flash memory allows the use of flash memory as non-volatile storage for the data initializer values. At reset, the values are copied to the VMA and the program begins execution with a known state for its initialized variables. The VMA for the `.data` section must be placed in RAM since a program may certainly modify the values of its initialized variables. The linker conveniently provides the built-in functions, `ADDR()` to obtain the VMA of a section and `LOADADDR()` to obtain the LMA of a section. So, for the copy table, the LMA of the `.data` section is the source for the copy and the VMA of the `.data` section is the destination. How to specify the LMA of a section to be different from its VMA is shown below.

The access to the linker generated table is completed by an external declaration of the the copy table giving the compiler the data type whose structure has been constructed to match the way in which the linker placed the values in memory.

Data copy table declaration

```
<<linker symbols: external declarations>>=
extern CopySegmentDesc const __data_copy_table_start__ ;
extern CopySegmentDesc const __data_copy_table_end__ ;
```

The initialized data section which survives reset still needs have its initializers present so they may be copied into RAM when the start up code detects that there has been a power on reset. At power on reset, the memory is indeterminate and the data initialization values must be copied to the correct location.

Noinit_data initialization table

```
<<linker tables: noinit data copy>>=
 __noinit_data_copy_table_start__ = . ;

LONG(LOADADDR(.noinit_data))
LONG(ADDR(.noinit_data))
LONG(SIZEOF(.noinit_data) / 4)

 __noinit_data_copy_table_end__ = . ;
```

```
<<linker symbols: external declarations>>=
extern CopySegmentDesc const __noinit_data_copy_table_start__ ;
extern CopySegmentDesc const __noinit_data_copy_table_end__ ;
```

The remaining memory areas which require initialization do *not* require a stored set of values. The memory values are being set to a fixed number, so the address of the section and its length are all that is needed. All of these sections must be allocated to RAM since the program writes to the memory during initialization and during execution.

Start with the `.bss` section.

BSS initialization table

```
<<linker tables: bss zero>>=
 __zero_table_start__ = . ;

LONG(ADDR(.bss))
```

[6] The term is slightly confusing since there is no *virtual* memory for this microcontroller as there might be for a processor with a Memory Management Unit (MMU). But a MMU serves to make some place in memory appear to the program as if it a physical address.

```
LONG(SIZEOF(.bss) / 4)

 __zero_table_end__ = . ;
```

And the corresponding external declarations for the code are:

```
<<linker symbols: external declarations>>=
extern InitSegmentDesc const __zero_table_start__ ;
extern InitSegmentDesc const __zero_table_end__ ;
```

The remaining initialization tables follow the established pattern.

Stack initialization table

```
<<linker tables: stack>>=
 __stack_table_start__ = . ;

LONG(ADDR(.process_stack))
LONG(SIZEOF(.process_stack) / 4)

LONG(ADDR(.main_stack))
LONG(SIZEOF(.main_stack) / 4)

 __stack_table_end__ = . ;
```

```
<<linker symbols: external declarations>>=
extern InitSegmentDesc const __stack_table_start__ ;
extern InitSegmentDesc const __stack_table_end__ ;
```

Noinit_bss initialization table

```
<<linker tables: noinit bss zero>>=
 __noinit_bss_table_start__ = . ;

LONG(ADDR(.noinit_bss))
LONG(SIZEOF(.noinit_bss) / 4)

 __noinit_bss_table_end__ = . ;
```

```
<<linker symbols: external declarations>>=
extern InitSegmentDesc const __noinit_bss_table_start__ ;
extern InitSegmentDesc const __noinit_bss_table_end__ ;
```

An external declaration of the `noinit_status` variable, used as part of the strategy for preserving some memory over a reset, is included. It's not really a linker generated symbol, but this is a convenient place to put it.

```
<<linker symbols: external declarations>>=
extern NoinitSegmentStatus noinit_status ;
```

With all the copy table descriptors defined, placing sections into the output can be continued.

ARM Exception Information

As part of the ARM ABI, some object files use exception unwinding information. This sometimes even appears in "C" code when it is intended to be used by "C++" code. The amount of space used is negligible.

```
<<linker section: ARM.exidx>>=
.ARM.extab : ALIGN(4)
{
    *(.ARM.extab* .gnu.linkonce.armextab.*)
```

```
} > FLASH

. = ALIGN(4) ;
PROVIDE(__exidx_start = .) ;
.ARM.exidx : ALIGN(4)
{
    *(.ARM.exidx* .gnu.linkonce.armexidx.*)
} > FLASH
PROVIDE(__exidx_end = .) ;                      /* ❶ */

__data_lma__ = . ;                              /* ❷ */
```

❶ *N.B.* the placement of these symbol definitions is slightly different from other symbols used to denote addresses within linker sections. The reason is to avoid a warning the linker will issue if the symbols are defined inside the section, but the content placed in the section itself defines no symbols. Normally, the linker sets the `sh_link` field, but if there are symbols defined inside the output section, the warning, `warning: sh_link not set for section .ARM.exidx`, is issued. One could argue that the linker should handle this case. You prefer *not* to see the warning as it is not warning us of anything useful.

❷ This symbol indicates the location of the load address for the `.data` section. It is placed here because this is the last section defined to contain data which is read only. Should other sections of read only data be required, the symbol might have to be moved. The intent here is to load the `.data` section immediately after the `.ARM.exidx` section.

Initialized data section

Previously, the difference between the VMA and LMA for a section was discussed. The `.data` section is distinctive because the section data is loaded at one address and linked to be run at a different address. The linker `AT` operation sets the LMA for a section.

```
<<linker section: data>>=
.data : AT (__data_lma__) ALIGN(4)
{
    __data_start__ = . ;

    *(.data*)

    . = ALIGN(4) ;
    __data_end__ = . ;
} > RAM
```

The previous script fragment uses the `AT` operation to set the LMA of the `.data` section to be at the value of the `__data_lma__` variable, but the VMA is placed in RAM.

```
<<linker symbols: external declarations>>=
extern uint32_t __data_start__ ;
extern uint32_t __data_end__ ;
```

Stack section

The first RAM section is for the Process Stack. A significant difference for the stack section is the need to align it on a 64-bit boundary for AAPCS reasons. The stack is just an area of memory, contains no data, and is initialized at start up, the `NOLOAD` section type is used to inform the linker there is nothing to place into the executable file.

```
<<linker section: process stack>>=
.process_stack (NOLOAD) : ALIGN(8)
{
    __process_stack_limit__ = . ;
```

```
    KEEP(*(.process_stack*))

    . = ALIGN(8) ;
    __process_stack_top__ = . ;
} > RAM
```

The Main Stack is placed at the next RAM location.

```
<<linker section: main stack>>=
.main_stack (NOLOAD) : ALIGN(8)
{
    __main_stack_limit__ = . ;

    KEEP(*(.main_stack*))

    . = ALIGN(8) ;
    __main_stack_top__ = . ;
} > RAM
```

Both the Main and Process stacks are placed in the first locations of RAM for the following reasons:

- Since the stack grows down, should stack activity dip below the allocated space, there is no memory there and the SOC generates a Hard Fault exception. This is preferable to overwriting some other portion of RAM. Since the Process Stack is more difficult to estimate, it is placed first with the Main Stack following.
- The first 64 KiB of RAM are TCM RAM. Since stack areas are accessed frequently, it is advantageous to place the stack in TCM RAM.

```
<<linker symbols: external declarations>>=
extern uint64_t __main_stack_limit__ ;
extern uint64_t __main_stack_top__ ;
extern uint64_t __process_stack_limit__ ;
extern uint64_t __process_stack_top__ ;
```

Note the external declarations are to 64-bit quantities in keeping with the 8 byte alignment for the section.

Uninitialized data section

The `.bss` section is similar to the stack. There is nothing to place in the executable so it is designated as a `NOLOAD` section. Unlike the stack, there are variables allocated by the program code in the `.bss` section and the linker must resolve those references in the code. It is also traditional to place the `COMMON` data in the `.bss` section. Common data is an historical remnant of past programming practices, but it is conventional to include it in the linker script.

```
<<linker section: bss>>=
.bss (NOLOAD) : ALIGN(4)
{
    __bss_start__ = . ;

    *(.bss*)
    *(COMMON)

    . = ALIGN(4) ;
    __bss_end__ = . ;
} > RAM
```

```
<<linker symbols: external declarations>>=
extern uint32_t __bss_start__ ;
extern uint32_t __bss_end__ ;
```

Non-initialized data section

After placing the `.bss` section in RAM, the same type of placement is used for the `.noinit_bss` section. This section is zeroed by the start up code only upon power on reset.

```
<<linker section: noinit data>>=
.noinit_data : AT (LOADADDR(.data) + SIZEOF(.data)) ALIGN(4)
{
    __noinit_data_start__ = . ;

    *(.noinit_data*)

    . = ALIGN(4) ;
    __noinit_data_end__ = . ;
} > RAM
```

Linker symbols are necessary to support the copying of initializer values. Note that unlike the initializer copying for the `.data` section, only a single block of `.noinit_data` section initializers is supported.

```
<<linker symbols: external declarations>>=
extern uint32_t __noinit_data_start__ ;
extern uint32_t __noinit_data_end__ ;
```

Non-initialized bss section

The final RAM section is similar to `.bss`. The difference, as seen previously, is how the area is initialized at start up allowing values to be preserved across system reset.

```
<<linker section: noinit bss>>=
.noinit_bss (NOLOAD) : ALIGN(4)
{
    __noinit_bss_start__ = . ;

    PROVIDE(noinit_status = .) ;
    . += 8 ;                    /* 8 == sizeof(NoinitSegmentStatus) */

    *(.noinit_bss*)

    . = ALIGN(4) ;
    __noinit_bss_end__ = . ;
} > RAM
```

```
<<linker symbols: external declarations>>=
extern uint32_t __noinit_bss_start__ ;
extern uint32_t __noinit_bss_end__ ;
```

Linker symbols header file

The literate program chunks for the linker symbol declarations are collected into a header file. This is simply a convenience for those parts of the code which make use of the symbols.

```
<<linker_symbols.h>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
```

```
 * Module:
 *   External declarations for linker generated symbols.
 *--
 */
#ifndef LINKER_SYMBOLS_H_
#define LINKER_SYMBOLS_H_

/*
 * Include files
 */
#include <stdint.h>
/*
 * Data type declarations
 */
<<linker symbols: data type declarations>>
/*
 * External Declarations
 */
<<linker symbols: external declarations>>

#endif /* LINKER_SYMBOLS_H_ */
```

`main` at last

Finally, a trivial program to resolve the `main` symbol can be written.

```
<<path-to-main-test.c>>=
<<edit warning>>
<<copyright info>>
#include <stdlib.h>

void
main(
    int argc,
    char **argv)
{
}
```

Building

The full build process is contained in a `Makefile` and involves building the startup code libraries. The linking stage of the program is done using `gcc` as a *driver* and passing the multitude of command line options required.

The `readelf` program can print a report of the resulting executable file.

```
There are 26 section headers, starting at offset 0x4b9bc:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00010000 010000 0005d0 00  AX  0   0  4
  [ 2] .build_id         NOTE            000105d0 0105d0 000024 00   A  0   0  4
  [ 3] .rodata           PROGBITS        000105f4 0105f4 000038 00  WA  0   0  4
  [ 4] .data             PROGBITS        10002000 012000 000068 00  WA  0   0  4
  [ 5] .bss              NOBITS          10002068 012068 00000c 00  WA  0   0  4
  [ 6] .process_stack    NOBITS          10000000 020000 001000 00  WA  0   0  8
  [ 7] .main_stack       NOBITS          10001000 020000 001000 00  WA  0   0  8
  [ 8] .noinit_data      PROGBITS        10002074 012068 000000 00   W  0   0  4
```

```
  [ 9] .noinit_bss       NOBITS          10002074 012068 000008 00  WA  0   0  4
  [10] .debug_info       PROGBITS        00000000 012068 002b9b 00      0   0  1
  [11] .debug_abbrev     PROGBITS        00000000 014c03 0007e5 00      0   0  1
  [12] .debug_aranges    PROGBITS        00000000 0153e8 000080 00      0   0  1
  [13] .debug_ranges     PROGBITS        00000000 015468 000220 00      0   0  1
  [14] .debug_macro      PROGBITS        00000000 015688 0093c9 00      0   0  1
  [15] .debug_line       PROGBITS        00000000 01ea51 001404 00      0   0  1
  [16] .debug_str        PROGBITS        00000000 01fe55 029730 01  MS  0   0  1
  [17] .comment          PROGBITS        00000000 049585 000049 01  MS  0   0  1
  [18] .ARM.attributes   ARM_ATTRIBUTES  00000000 0495ce 000034 00      0   0  1
  [19] .debug_frame      PROGBITS        00000000 049604 000328 00      0   0  4
  [20] .debug_loc        PROGBITS        00000000 04992c 000d09 00      0   0  1
  [21] .stab             PROGBITS        00000000 04a638 00003c 0c     22   0  4
  [22] .stabstr          STRTAB          00000000 04a674 000076 00      0   0  1
  [23] .symtab           SYMTAB          00000000 04a6ec 000ac0 10     24  77  4
  [24] .strtab           STRTAB          00000000 04b1ac 000703 00      0   0  1
  [25] .shstrtab         STRTAB          00000000 04b8af 00010d 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
```

Testing

As a first step in starting the processor, manual testing and observation is necessary. At this point, there is no infrastructure in place that would allow reporting the test results and, consequently, no means to make an evaluation of whether the actual results matched the expected results.

Some screen shots show what the debugger sees in memory. The following figure shows the values placed in the memory by the startup code.

- The exception vector table is as expected. The first entry is the top of the stack. Next comes the Reset Handler. The remaining exceptions all point to the Default Handler.
- The two stack areas have been initialized with an end marker and a repeating pattern.
- The value of the `SystemCoreClock` is 48 MHz, as expected.

Global Data

Name	Value	Location	Size	Type	Scope
__Vectors		0001 0000	192	const void()*[48]	startup_apollo3.c
[0]	1000 2000	0001 0000	4	void()*	startup_apollo3.c
[1]	Reset_Handler (0001 00E1)	0001 0004	4	void()*	startup_apollo3.c
[2]	Default_Handler (0001 00C5)	0001 0008	4	void()*	startup_apollo3.c
[3]	Default_Handler (0001 00C5)	0001 000C	4	void()*	startup_apollo3.c
[4]	Default_Handler (0001 00C5)	0001 0010	4	void()*	startup_apollo3.c
[5]	Default_Handler (0001 00C5)	0001 0014	4	void()*	startup_apollo3.c
[6]	Default_Handler (0001 00C5)	0001 0018	4	void()*	startup_apollo3.c
[7]	Default_Handler (0001 00C5)	0001 001C	4	void()*	startup_apollo3.c
[8]	Default_Handler (0001 00C5)	0001 0020	4	void()*	startup_apollo3.c
[9]	Default_Handler (0001 00C5)	0001 0024	4	void()*	startup_apollo3.c
<member display limited by user preference>					
argc	0000 0000	1000 2008	4	int	startup_apollo3.c
argv		1000 2004	4	char*[1]	startup_apollo3.c
main_stack	"'\357''\276''\255''\336''\02	1000 1000	096	unsigned char[4096]	startup_apollo3.c
[0]	0xEF ('ï')	1000 1000	1	unsigned char	startup_apollo3.c
[1]	0xBE ('¾')	1000 1001	1	unsigned char	startup_apollo3.c
[2]	0xAD ('')	1000 1002	1	unsigned char	startup_apollo3.c
[3]	0xDE ('Þ')	1000 1003	1	unsigned char	startup_apollo3.c
[4]	0x12 ('\018')	1000 1004	1	unsigned char	startup_apollo3.c
[5]	0x34 ('4')	1000 1005	1	unsigned char	startup_apollo3.c
[6]	0x56 ('V')	1000 1006	1	unsigned char	startup_apollo3.c
[7]	0x78 ('x')	1000 1007	1	unsigned char	startup_apollo3.c
[8]	0x12 ('\018')	1000 1008	1	unsigned char	startup_apollo3.c
[9]	0x34 ('4')	1000 1009	1	unsigned char	startup_apollo3.c
<member display limited by user preference>					
process_stack	"'\357''\276''\255''\336''\02	1000 0000	096	unsigned char[4096]	startup_apollo3.c
[0]	0xEF ('ï')	1000 0000	1	unsigned char	startup_apollo3.c
[1]	0xBE ('¾')	1000 0001	1	unsigned char	startup_apollo3.c
[2]	0xAD ('')	1000 0002	1	unsigned char	startup_apollo3.c
[3]	0xDE ('Þ')	1000 0003	1	unsigned char	startup_apollo3.c
[4]	0x12 ('\018')	1000 0004	1	unsigned char	startup_apollo3.c
[5]	0x34 ('4')	1000 0005	1	unsigned char	startup_apollo3.c
[6]	0x56 ('V')	1000 0006	1	unsigned char	startup_apollo3.c
[7]	0x78 ('x')	1000 0007	1	unsigned char	startup_apollo3.c
[8]	0x12 ('\018')	1000 0008	1	unsigned char	startup_apollo3.c
[9]	0x34 ('4')	1000 0009	1	unsigned char	startup_apollo3.c
<member display limited by user preference>					
SystemCoreClock	48 000 000	1000 2000	4	unsigned long	system_apollo3.c

Figure 2.4: Data Values after Startup

The main program has no statements. If execution continues, returning from main, the startup code invokes `abort()`. Since a debugger is present, the execution state can be captured and is shown below.

```
69  extern void main(int argc, char **argv) ;
70  /*
71   * External Functions
72   */
73  noreturn void
74  start_main(void)
75  {
76      uint32_t control = __get_CONTROL() ;                // <1>
77      control = BTWD_FIELD_SET(control, CONTROL_SPSEL) ;
78      control = BTWD_FIELD_SET(control, CONTROL_nPRIV) ;
79      __set_CONTROL(control) ;
80      __ISB() ;
81
82      main(argc, argv) ;
83
84      abort() ;                                           // <2>
85  }
86
```

Figure 2.5: Returning from main

Summary

In this chapter, the executable code and scripting necessary to get an Apollo 3 SOC to initialize an execution environment so that an application can begin at `main()` has been shown. This is certainly not the only way to accomplish this goal. There are a large number of ways to get to `main()` and the implementation of a startup sequence, like anything else in software, needs to meet the requirements of the target system.

This design has several limitations which more experienced readers will have noticed.

- There is no support for `C++` programs. The `C++` language imposes additional requirements which must be accomplished at initialization time and additional start up code and linker sections are needed. Since there is no intention to include any `C++` code for this project, the startup sequence was simplified.
- The GNU ARM toolchain includes the `newlib` implementation of the standard "C" library. That library contains startup code which can handle a wider variety of language requirements, but has no system specific features such as those implemented here, *e.g.* a `.noinit_bss` section.
- There is no support in this startup code for handling the requirements of the `atexit()` library function. This function allows for registering one or more functions to be invoked when a program terminates. Since the class of application programs targeted here do not terminate, this feature was excluded. Further, there is no support for GCC "C" language extensions such as constructors or destructors. Using GCC attribute extensions, functions may be given the `constructor` attribute and they are invoked before main or the `destructor` attribute and they are invoked after main has completed or upon exit. These can be used as hooks to provide system dependent behavior, but no support for executing them is provided by this startup code.
- Finally, experienced readers will have noticed that there is no provision for a system wide heap for dynamic memory allocation. The use of system heaps in microcontroller based systems is a subject that has generated considerable discussion in the greater microcontroller community. Some hold the opinion that it should not be used at all. There are good reasons, such as fragmentation and recovery potential, not to use a system-wide heap. Others hold that heap allocation during initialization is acceptable but not afterward. Still others see no problems using a system-wide heap. For the type of applications targeted here, *e.g.* those which are long running and have deterministic requirements, no system wide heap is used and its negative aspects are avoided. When there are specific situations that require dynamic memory allocation, that allocation is managed with worst case, compile-time-allocated memory blocks tied to the data type of the allocated memory objects.

Chapter 3

From `main` to "hello, world"

The previous chapter brought you to the point of powering up the processor and initializing memory so that control can be transferred to the `main()` function. In this chapter, terminal output is added. The output functionality is provided by the Segger® RTT scheme.

The Required First Program

Ever since Kernighan and Ritchie published *The "C" Programming Language*, it has become *de rigueur* to provide a simple example to print to a terminal as the starting point for explaining new languages and systems. The simplicity of the classic "hello, world" program belies the amount of system infrastructure which must be in place for it to work. Kernighan and Richie admit to the need to overcome the mechanics of creating the program text file and compiling the program to yield an executable. But there is an implicit assumption that this first programming activity takes place in the environment of an operating system. This is a natural assumption for most circumstances, but not for a microcontroller which has just powered up.

When coming from the bottom up, you must undertake providing the required I/O infrastructure. "C" delegates the details of actually producing formatted output to library functions. This makes it possible to divorce the programming language from the operating environment so that our own implementation of the library may be substituted to suit the particulars of the environment.

"C" Standard Libraries

"C" does not have the extensive standard libraries that accompany more recent languages. Most of the functions in the standard library support only basic operations. Operations on more complex data structures or even robust string handling, are missing.

There are several implementations of the standard "C" library which might suit our needs. This design uses `newlib` for these reasons.

- The library comes integrated with the GNU Toolchain for ARM processors. This saves the effort to build an install the library.
- The `newlib` library is complete and well maintained.
- The library can be staged on a new platform easily and includes the necessary stub functions for the system dependencies.
- The `nano` version of the library makes trade-offs between functionality and size by removing lesser used features, particularly reducing the size required for `printf()`.

The manner in which `newlib` integrates into a system is to assume an implementation of a set of *system calls* exists. A set of essentially non-functioning system calls is provided by the `nosys` library meet our purposes. But custom versions of system level I/O functions must be substituted in place of the stubs provided by `nosys`. The `nosys` library stubs are, with one exception, truly non-functional and generally return an error code if invoked. The individual functions are easily replaced by simply providing an implementation and insuring it is presented to the linker before `nosys`. To obtain terminal output, it is necessary to replace the `_write()` function.

I/O Requirements

Before launching off into implementing an I/O scheme for the platform, it is worthwhile to consider what types of I/O are needed. The primary need for is for terminal output which can be used for system introspection. Building interactive console applications is not a significant use case. But visibility into the execution of the system is essential. That visibility can be obtained in a number of ways.

- A physical UART device is commonly available on microcontroller SOC's. Serial UART technology existed before computers were invented and has survived to become the lowest common denominator of interfaces. Although the UART I/O may be intended for a human, frequently, UART's are used to communicate with more complicated peripherals such as cellular modems. In a later chapter, device code to run the Apollo 3 UART is given.
- The Cortex-M4 has instrumentation and tracing capability which can be output via the SWO pin. Unfortunately, the SWO pin on the Apollo 3 is not connected to the debugger connector on the SparkFun MicroMod processor board.
- Since a SEGGER J-Link device is used, the SEGGER RTT® (real time trace) is available. This method operates by including a small piece of code in the program and invoking various provided output functions.

For these purposes, the SEGGER RTT approach works well. It is much faster than a physical UART and much less intrusive on the execution load of the system. The SWO approach is somewhat slower than RTT and to *bug wire* the SWO pin over to the debugger connector is an unwelcome complication. Use of RTT also sets up for other SEGGER components that might be useful in the future.

The details of the RTT code are not described here as it is readily available on the SEGGER wiki. The integration into the system is provided by two source files. One is the RTT code proper and the other is an implementation of the `_write()` function which replaces the version from the `nosys` library. By compiling the files and insuring that the replacement version of `_write()` is linked in first, the RTT I/O is integrated with `newlib` and the Ozone debugger can directly capture and display the output. SEGGER provides several other means to view the RTT output.

A Version of "hello, world"

This version of the classic hello world program, starts along familiar lines.

```
<<hello world: main>>=
<<hello world: include files>>

void
main(void)
{
    <<hello world: I/O init>>

    printf("hello, world\n") ;

    <<hello world: report system information>>
}
```

As usual, a function prototype for `printf()` is needed from `stdio.h`.

```
<<hello world: include files>>=
#include <stdio.h>
```

None of the initialization performed in the last chapter involved peripheral devices or other application specific uses. To use RTT the `SEGGER_RTT_Init()` must be invoked function. The simplest approach is to invoke the function at the top of `main`. When the amount of application specific initialization becomes larger, there are better ways to organize it.

To allow removing the SEGGER RTT mechanism, the pre-processor symbol `USE_SEGGER_RTT` is defined.

```
<<hello world: I/O init>>=
#ifdef USE_SEGGER_RTT
SEGGER_RTT_Init() ;
#endif /* USE_SEGGER_RTT */
```

Function prototypes for the RTT functions are required.

```
<<hello world: include files>>=
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
```

System Information

Being able to print *"hello, world"* is an accomplishment, but there is much more information about the running system you would like to have available. Microcontroller SOC's have many registers which hold information about their identity and capablity. In this section, some of the contents available system information is printed. There's a lot more information available than printed here.

Some support functions to handle the details of formatting the system information are the mechanism for accessing the information.

```
<<hello world: include files>>=
#include "sysinfo.h"
```

The functions in this module have similar programming interfaces which are patterned after `snprintf()`. That's not surprising, since `snprintf()` is the primary function used to perform the output formatting. The system information functions accept as parameters a pointer to a character array where the output is placed and the size in bytes of the output array. The return value is an integer where:

- negative numbers indicate an error,
- positive numbers indicate the number of bytes written to the output array, exclusive of the terminating `NUL` character,
- overflow of the output space is indicated by returning the number of characters that would have been generated had there been sufficient space in the output array (again excluding the terminating `NUL`). If the returned value is greater than or equal to the value of the input size parameter, then the input has been truncated, but it is still properly `NUL` terminated.

That set of semantics is somewhat complicated, but works reasonably well in practice.

To print the information a common buffer is reused. The appropriate size is a guess, but the formatting functions do not overflow the buffer given to them.

```
<<hello world: report system information>>=
char info_string[256] ;
```

A note about variable length arrays

"C" variable length arrays that were introduced with `C99` are used in this design. Some developers hold the opinion that this construct should not be used in a microcontroller based system. While it is true that stacks can be made to overflow using a VLA and there are probably many abuses of the construct, there is no reason to avoid them when reasonable bounds can be placed on the array size. Stack overflow barriers are in place as discussed in the last chapter, so it is possible to detect and diagnose poor usage of the VLA feature during testing. Prudent use of a VLA can simplify certain code and they are useful for more than automatic memory allocation.

Printing the Build ID

From the previous work shown in the last chapter, you can now obtain the build ID from the executable file by providing a means to format it into a printable string.

```
<<hello world: report system information>>=
int formatted = sysi_format_build_id(sizeof(info_string), info_string) ;
if (formatted > 0 && formatted < sizeof(info_string)) {
    printf("Build ID: %s\n", info_string) ;
} else {
    puts("Build ID: formatting error") ;
}
```

Printing the Chip ID

Each Apollo 3 chip has a unique 64-bit identifier available in a register.

```
<<hello world: report system information>>=
formatted = sysi_format_chip_id(sizeof(info_string), info_string) ;
if (formatted > 0 && formatted < sizeof(info_string)) {
    printf("Chip ID: %s\n", info_string) ;
} else {
    puts("Chip ID: formatting error") ;
}
```

Printing the Chip Information

Embedded in the “part number” register are a set of bit fields which describe the basic features of the chip.

```
<<hello world: report system information>>=
formatted = sysi_format_chip_info(sizeof(info_string), info_string) ;
if (formatted > 0 && formatted < sizeof(info_string)) {
    puts(info_string) ;
} else {
    puts("Chip Info: formatting error") ;
}
```

Printing the Scratch Registers

As a final example, there are two “scratch” registers in the Apollo 3. The values of these registers are not affected by reset. Here they are read directly from memory and printed as hexadecimal values.

Access to the scratch registers needs the `apollo3.h` header file to know the register addresses. The formatting strings for 32-bit unsigned values which appropriate for our platform are contained in a standard header file.

```
<<hello world: include files>>=
#include "apollo3.h"
#include <inttypes.h>
```

```
<<hello world: report system information>>=
printf("Scratch Register 0: %08" PRIx32 "\n", MCUCTRL->SCRATCH0) ;
printf("Scratch Register 1: %08" PRIx32 "\n", MCUCTRL->SCRATCH1) ;
```

`hello, world` at last

The final source code for the “hello, world” program is little more than `main()` decorated with supplemental comments.

```
<<main-to-hello-world-test.c>>=
<<edit warning>>
<<copyright info>>
<<hello world: main>>
```

Running on the test platform, this version of hello world produces the following output.

```
hello, world
Build ID: 6b83_5c6d_c71b_70d2_6865_6c6c_6f2c_2077_6f72_6c64
Chip ID: ce0983be_2a826f08
CHIPPN: 0x06672198
    PN: Apollo3, FLASHSIZE: 1M, SRAMSIZE: 384KB, MAJOR_REV: B, MINOR_REV: 0, PKG: BGA, PINS ↩
        : 81, TEMP: 0, QUAL: 0
Scratch Register 0: 00000000
Scratch Register 1: 00000000
```

Chapter 4

Crossing the Divide

A previous chapter discussed how running thread mode computations in unprivileged mode effectively erects a barrier between handler mode and thread mode. This chapter shows how foreground and background activities cooperate to transfer control and data between the two sides. This mechanism is central to the design of how the system responds to its environment.

Handler Mode / Thread Mode Split

Only handler mode is allowed[1] to access system and peripheral registers. The only way that handler mode is entered is by an exception.

This chapter shows the design elements needed to work with the split between privileged and unprivileged code. To be clear, the split between handler mode and thread mode is intended to divide the area of system concerns between those aspects which monitor and interact with the external environment from those which are associated with the logic of an application's response to the environment. Since the core architecture provides the mechanism to handle this split, minimal additional software execution is required to provide the separation.

Separating and confining different system concerns is another recurring topic in this book.

There are four mechanisms used to enter handler mode.

Interrupt requests
: Peripheral devices can cause can *pend* an interrupt. The processor core then arbitrates between pending interrupts to select the order in which they are made active.

System faults
: The system can detect certain faulty behavior of both the hardware and software to cause an exception.

Thread mode request
: The Cortex-M architecture provides the `SVC` instruction as the primary means for thread mode software to create an exception. The exception created by the `SVC` instruction is used to request services from handler mode software.

Handler mode request
: The other architectural exception mechanism provided is the `PendSV` exception. The intended use of this mechanism is to have handler mode software request additional handler mode services. It serves as a means for deferring computations which started in handler mode to a lower priority exception. The usual use case allows a higher priority interrupt to defer additional work to a lower priority. Continuing additional hander mode processing at a lower priority prevents blocking other, potentially high priority exceptions, any longer than is necessary. This mechanism is sometimes used for switching between virtual execution contexts (*e.g.* task context switching). This design does not the `PendSV` mechanism.

[1] More properly, the system is configured so that only handler mode can perform system and peripheral register access.

Since the goal is to build a *reactive* system, all actions of the system are ultimately initiated by handler mode activities. Even periodic time based actions are ultimately driven by an interrupt from a timer peripheral. This section gives the design of the how handler mode exception processing is managed. In another section, the manner in which thread mode computations are managed is shown.

Handler mode computation has the following properties.

- Handler mode is the only provided mechanism that can preempt thread mode activities. In keeping with a reactive systems focus, handler mode preemption allows the system to track and react to the environment at speeds governed by the environment itself. This design does *not* include any mechanism which allows thread mode computations to preempt other thread mode computations, *i.e.* this design does create any execution contexts beyond that provided by the processor core.
- Handler mode computation runs to completion and there is no waiting or pausing. Computations which requires synchronization between different parts of the software are run in thread mode.
- Handler mode notifies thread mode activities of the occurrence of changes in the environment and can *push* data and control to thread mode activities.
- Conversely, thread mode activities can only affect the system environment by making requests to handler mode actions which ultimately involve peripheral control with the possibility of *pushing* data and control into the environment.

Exception Priorities

It is a goal of this design to manage the order and sequencing of computation in the software strictly by manipulating the execution priority of the software components using the mechanisms provided by the core to configure execution priority. By using the mechanisms directly supported by the core, the amount of additional software execution required just for deciding which code is to run next is lessened. In this section, the priorities assigned to the various architectural mechanisms provided by the ARMv7-M architecture are described. The Apollo 3 Blue SOC provides 3-bits of exception priority, as is quite common and is the minimum defined by the core architecture. Three bits equals 8 priority levels. Exception priorities are discussed in terms of these 8 levels but the same design considerations apply independent of the number of priority levels. Staging the discussion in terms of the specifics of an 8 level scheme makes more concrete the choice of design elements supported by the core which match the system goals.

As a further simplification, this design does *not* use *sub-priorities*. Sub-priorities direct the choice of which exception to run when there are multiple pending exceptions at the same *priority*. Without sub-priorities, the processor chooses the smallest IRQ number from among simultaneously pending requests which have the same priority. The mechanisms provided by the ARMv7-M architecture are intended to support a large number of different system design goals. Here, there are specific requirements that can be met using a less complicated arrangement of core features.

The following describes the rationale for the assignment of exception priorities.

- All system faults which have a configurable priority run at the highest priority. If the core detects a problem, it is desirable to act upon it at once. The Reset, NMI, and Hard Fault exceptions have assigned, fixed priority that are always higher than any configurable priority.
- Provisions are made for a debug monitor priority which is less than a system fault. An *in-system* debug monitor has a number of uses.
- All interrupt requests happen at the same priority. This is a *flat* interrupt priority scheme. Although ARMv7-M core supports using priority to *nest* interrupt service, that complication is avoided in this design. As shown later, the assignment of priorities in this design allows for a use case with nested of interrupts, but that capability is not used here based on complexity arguments alone.
- The `SVC` priority is lower than that of an interrupt request and is the lowest priority exception considered. This choice means that IRQ's can preempt the SVC handler. This requires taking more care in dealing with peripheral device access, but it implies that interacting with the environment has higher priority over internal system execution.

The following table shows the prioritization scheme. To reiterate, a lower priority level preempts a higher priority level and the priority number accounts for the number of priority bits supported by the core and the location of the bits in the system registers.

Table 4.1: Exception Priority Assignments

Priority level	Priority number	Description
0	0x00	All configurable system fault handlers.
1	0x20	Unused.
2	0x40	Reserved for debug monitor.
3	0x60	Unused.
4	0x80	Unused.
5	0xA0	All peripheral device interrupts.
6	0xC0	`SVC` exception.
7	0xE0	Unused.

The following priority levels are not used by this design, but some possible use cases are described.

Priority 1 — can be used if an interrupt is needed to work in conjunction with the debug monitor. For example, if the debug monitor is receiving input from a UART, it may be more convenient to handle the UART with interrupts and those interrupts would need to have a higher priority than the debug monitor. Alternatively, it may be necessary for some peripheral devices to continue to be serviced even when a debug monitor is running.

Priority 3 - 4 — can be used to implement a nested interrupt scheme for those systems which have requirements where a flat scheme may not be sufficient. Note that nested interrupt schemes require careful design to get correct and make the system more difficult to reason about. Also note that there are specific code sequences given later which depend upon flat IRQ prioritization. Those situations are pointed out as they arise.

Priority 7 — can be used for the `PendSV` exception. This design does not use `PendSV`, but it might be needed by systems which have a more complicated nested interrupt structure and which might wish to defer some of the actions for a particular interrupt to a lower exception priority.

Initializing Exception Priorities

From the above discussion, the priorities can be encoded as:

```
<<exc priority: constants>>=
#define PRIORITY_GROUP              4
#define SYS_HANDLER_PRIORITY        0
#define DEBUG_MONITOR_PRIORITY      2
#define IRQ_PRIORITY                5
#define SVC_PRIORITY                6
```

In Chapter 2, a *weak* function was defined to initialize the exception priorities. Now that function can be replaced with code that assigns priorities according to the above specifications.

SystemExcPriorityInit Implementation

```
<<system exc priority init: external function definitions>>=
void
SystemExcPriorityInit(void)
{
    NVIC_SetPriorityGrouping(PRIORITY_GROUP) ;
    uint32_t const sys_handler_priority_number =
            NVIC_EncodePriority(PRIORITY_GROUP, SYS_HANDLER_PRIORITY, 0) ;  // ❶
    NVIC_SetPriority(MemoryManagement_IRQn, sys_handler_priority_number) ;
    NVIC_SetPriority(BusFault_IRQn, sys_handler_priority_number) ;
```

```
    NVIC_SetPriority(UsageFault_IRQn, sys_handler_priority_number) ;

    uint32_t const debug_monitor_priority_number =
            NVIC_EncodePriority(PRIORITY_GROUP, DEBUG_MONITOR_PRIORITY, 0) ;
    NVIC_SetPriority(DebugMonitor_IRQn, debug_monitor_priority_number) ;

    uint32_t const irq_priority_number =
            NVIC_EncodePriority(PRIORITY_GROUP, IRQ_PRIORITY, 0) ;
    for (IRQn_Type irqn = BROWNOUT_IRQn ; irqn <= CLKGEN_IRQn ; irqn++) {
        NVIC_SetPriority(irqn, irq_priority_number) ;                          // ❷
    }

    uint32_t const svc_priority_number =
            NVIC_EncodePriority(PRIORITY_GROUP, SVC_PRIORITY, 0) ;
    NVIC_SetPriority(SVCall_IRQn, svc_priority_number) ;

    stwd_enable_bus_fault(true) ;                                              // ❸
    stwd_enable_usage_fault(true) ;
    dtwd_enable_fault_capture(true) ;
}
```

❶ The `NVIC_EncodePriority()` macro produces the proper number to write to system registers for the given priority. Recall that sub-priorities are not used.

❷ By setting all the peripheral interrupts to the same level, any missing peripheral device interrupts that are inadvertently triggered cause the `SystemMissingException()` function to be executed at the same priority as all the other IRQ's.

❸ Enable all the fault related exceptions and the Apollo 3 specific fault capture mechanism.

Foreground / Background Interfaces

In the last section, the exception prioritization scheme to order preemptive execution was described. Exception processing was discussed using ARMv7-M architectural terms such as Thread Mode and Handler Mode. From here onward, the older more generic terms of *foreground* and *background* are used. For the ARMv7-M architecture, *foreground* execution is defined as the processing which happens in Handler Mode, and *background* execution as the processing which happens in Thread Mode. Other processor architectures may not have the same configurability as the ARMv7-M architecture, but many of the same considerations apply, even when reduced to the simple two mode scheme of interrupt and non-interrupt execution where no priority configuration is possible.

This section describes how the division between foreground and background processing is coordinated. Design concepts are presented for how background processing can access privileged services and how foreground processing can notify background processing of happenings in the system environment.

Climbing Toward the Foreground

Since a barrier between the background and the foreground has been erected, it is necessary to design a mechanism to allow the two sides to interact. The design concept used is that of a *proxy*. When background processing needs to control devices or aspects of the system execution, it makes a request to the foreground to obtain the required service. Conceptually, the request is similar to a message. Fortunately, the mechanism operates within a single processor core, and simpler memory-based mechanisms can be used to deliver the requests.

A function executing in the background makes a request to a *proxy function* which executes in the foreground to fulfill the request. The proxy function can be viewed as a continuation of the background request which is executed with privilege. The `SVC` instruction is the core architectural mechanism used to enable the background request to obtain service from the foreground proxy. The following figure shows a simplified schematic of how background requests are routed to foreground functions acting a proxy for the request.

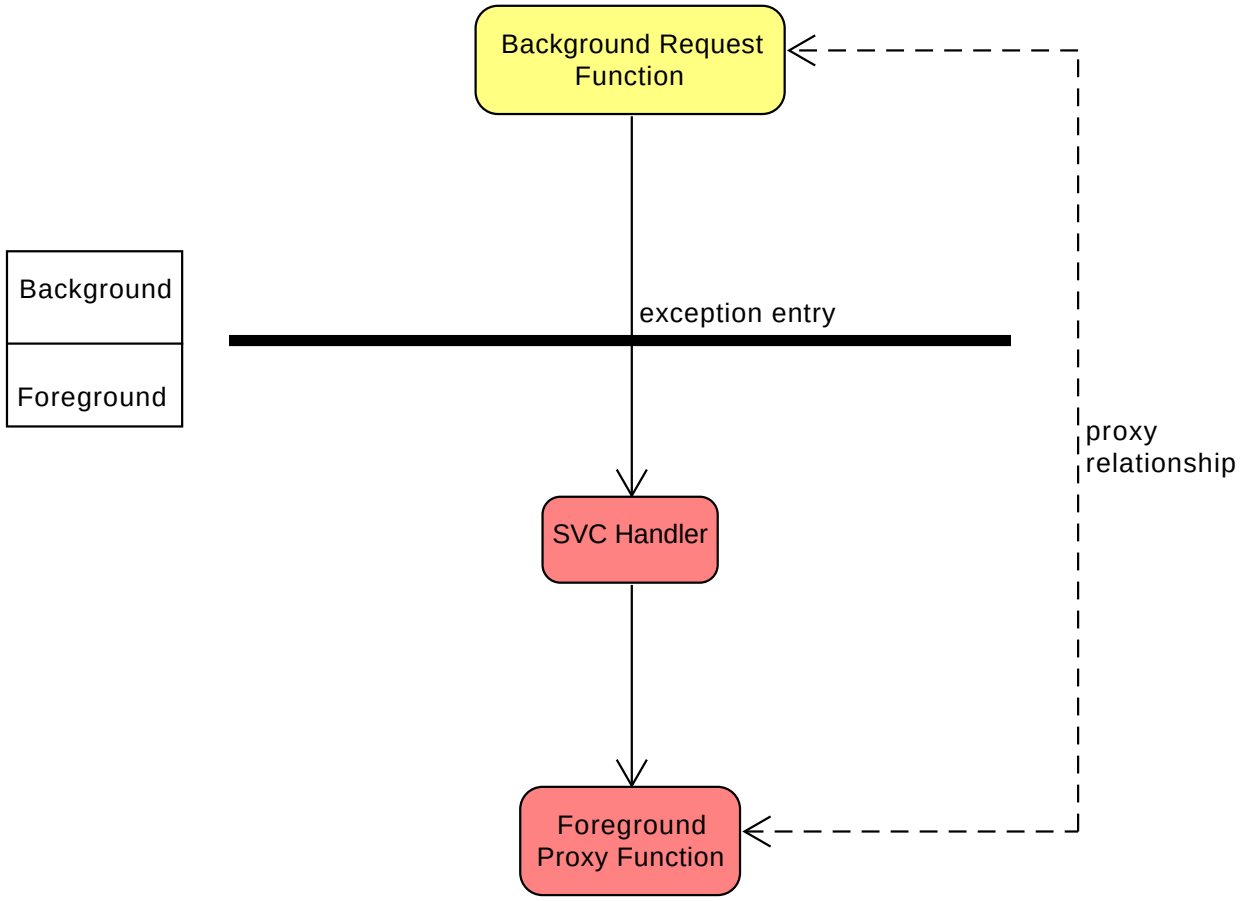

Figure 4.1: Overview of Proxy Relationship

In this figure, the yellow box represents a request from background processing for a service which requires privilege to execute. The red boxes represent foreground processing. Control is transferred from the **Background Request Function** to the **Foreground Proxy Function** by means of the `SVC` exception which is handled by the **SVC Handler**. There is a one-to-one correspondence between the **Background Request Function** and its **Foreground Proxy Function** counterpart.

The scheme shown in the previous figure is overly simplistic for realistic programs. The number of request functions is large considering that all peripheral device access and other "device driver" type operations must be performed in the foreground. Peripheral device control is a major part of any microcontroller system and an organization for how device control software is included needs to be designed. A long, unstructured enumeration of function names is not a sufficient organization for the foreground proxy functions. To make the groupings smaller, a hierarchy is used, as is common practice, and the background requests are divided into subcomponents related by a tree arrangement.

The first subdivision in the foreground organization is the concept of *service realms*. There are two service realms in this design.

1. System realm
2. Device realm

The boundary is somewhat arbitrary. Functionality to manage processor core resources and control execution sequencing is deemed part of the *system* realm. Additionally, the system realm does not have any asynchronous execution requirements. Asynchronous activity arising from problems detected by the processor core is delegated to system fault handlers and does not require any background execution to resolve. Since this design considers system faults as fundamentally unrecoverable, no mechanism is provided for the system realm to direct asynchronous background execution.

Functions associated with peripheral devices are deemed part of the *device* realm. In the device realm, the requests are partitioned first by *device class* and second by *device operation*. The class of a device is a logical grouping of one or more peripheral devices which act in concert together. The logical device supports a set of device operations, that are, in general, specific to the class. A device class may also have multiple *device instances* which all operate the same and according to the defined device operations. Some other terms used for device class and device instance are major device number and minor device number.

The following figure expands upon the previous figure by including the division into service realms and, for the device realm, the further subdivision into device classes and device operations. This gives a more accurate picture of the proxy relationship between background requests and foreground proxy functions when they involve different service realms.

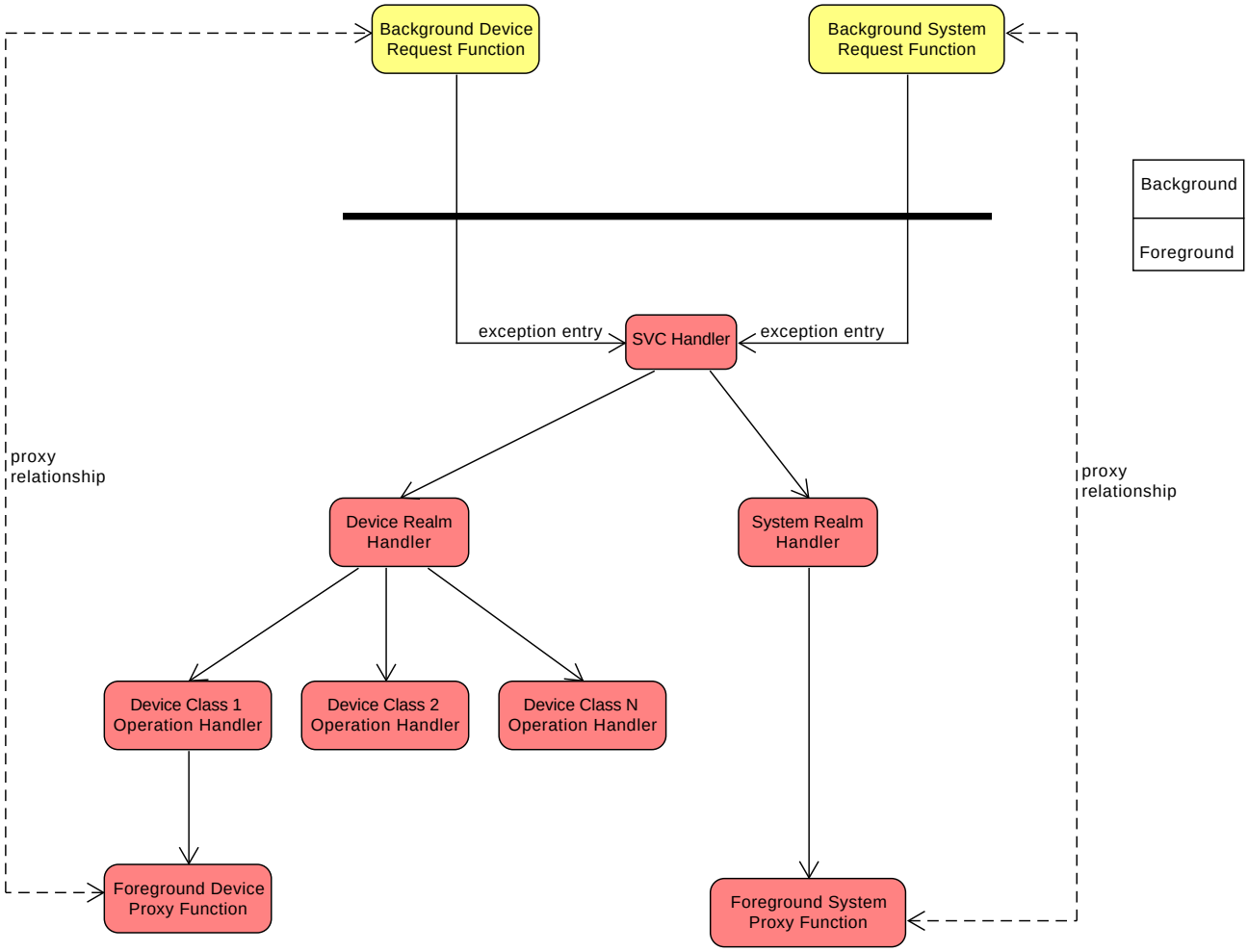

Figure 4.2: Proxy Relationship with Service Realms

In this figure, the two service realms have been added. The **SVC Handler** decides to which realm the request is directed. System realm requests are dispatched using only a single level of decision. Device realm requests are dispatched first to a particular device class and then, in a second level, the requested operation is dispatched to a proxy function. The device instance is passed along as a parameter to the proxy function so it may choose the particular hardware instance on which it operates.

Climbing Toward the Background

In the previous section, the design concepts for transferring control and data from background processing to foreground processing were discussed. It is necessary to provide mechanisms which traverse the barrier in the other direction. It would be

convenient if the traversal from foreground to background were symmetric with the background to foreground direction just shown. However, it is not.

Recall that the initiation of all execution in the system ultimately arises from an interrupt request (IRQ). In general, an IRQ handler needs to inform the background of the occurrence of the interrupt and often must transfer data from the system environment to the background for further action. Much of the requirements for IRQ handlers interacting with the background derive from the desire to minimize the amount of time spent at IRQ level in order to provide timely response to other happenings in the environment. This is part of the currency of the environment which must be handled by a reactive system.

This design is driven by changes in the environment which are transported into the system by interrupts. The processor core invokes the IRQ handler which is then responsible for starting the computations necessary to react to the environment. The required work to fully realize the effect of the interrupt is typically split between foreground processing and background processing. It is an important design consideration to determine how the required computations are split. Platitudes such as, "Interrupt handlers should be as short as possible," give no practical guidance for determining where the processing associated with an interrupt occurs.

At a minimum, an IRQ handler must:

- Re-arm the interrupt for the next occurrence. Interrupts are not a one-time happening. The source of the interrupt must be made quiescent so that the next time the interrupt is asserted the system will recognize another instance of the interrupt.
- Capture any state of the environment which may be present in the registers of a peripheral device. For example, a UART receive interrupt needs to read the received character out of the device. Otherwise, the UART receiver may overflow and any message composed of the received bytes would be corrupted.
- Move data or control into the environment. For a UART transmit interrupt, data from the system must be moved into the peripheral device registers for transmission.
- Recognize semantically significant conditions and initiate the processing to complete the required function. Consider the case where data is obtained from the environment in block units. When a complete block has been received, background processing must be notified that a received block is ready for the additional processing required to produce the system response.

Because the processor can only do one thing at a time and because while handling an IRQ nothing else can execute[2], to remain responsive to the environment, it is desirable to have the execution time of an IRQ handler be small. Of course, IRQ handlers always take some time to execute. It is important to recognize that the longest running IRQ handler determines the worst case response latency of the system for detecting changes in the environment.

The detailed manner in which a logical device operates and the manner in which an IRQ handler contributes to the overall device operation is an element of its design. There is considerable flexibility as to where processing is allocated and practical patterns have been established for where the split in processing occurs. However, it is a goal to have only one way to notify the background processing of conditions detected by the IRQ handler. To accomplish this goal, it is necessary to define a mechanism for IRQ handlers to generate background notifications that are used by background processing to compute the system response.

Because the environment of the system operates asynchronously to the program and because the timing of happenings in the environment cannot be predicted, a queuing mechanism is used to allow IRQ Handlers to submit notifications. Background processing in not synchronous to foreground interrupts. The following figure shows an overview of the background notification scheme.

Handler Mode Only Processing

For certain types of systems, typically small systems which act as a simple bridge between components, it is possible that all processing can be done in the foreground. This type of design eliminates any background processing and all IRQ handling runs to completion to generate the system response. It is a common enough design that the ARMv7-M architecture supports a special sleep mode. Setting the `SLEEPONEXIT` bit in the System Control Register causes the processor to enter sleep (or deep sleep) upon returning from an IRQ exception. Thus if the longest service time of an IRQ is within the bounds of the desired response latency of the application, it is possible to build a system composed entirely of IRQ handers.

[2]This is strictly true in a flat interrupt priority design. But even in a nested interrupt design, many interrupts run at the same priority.

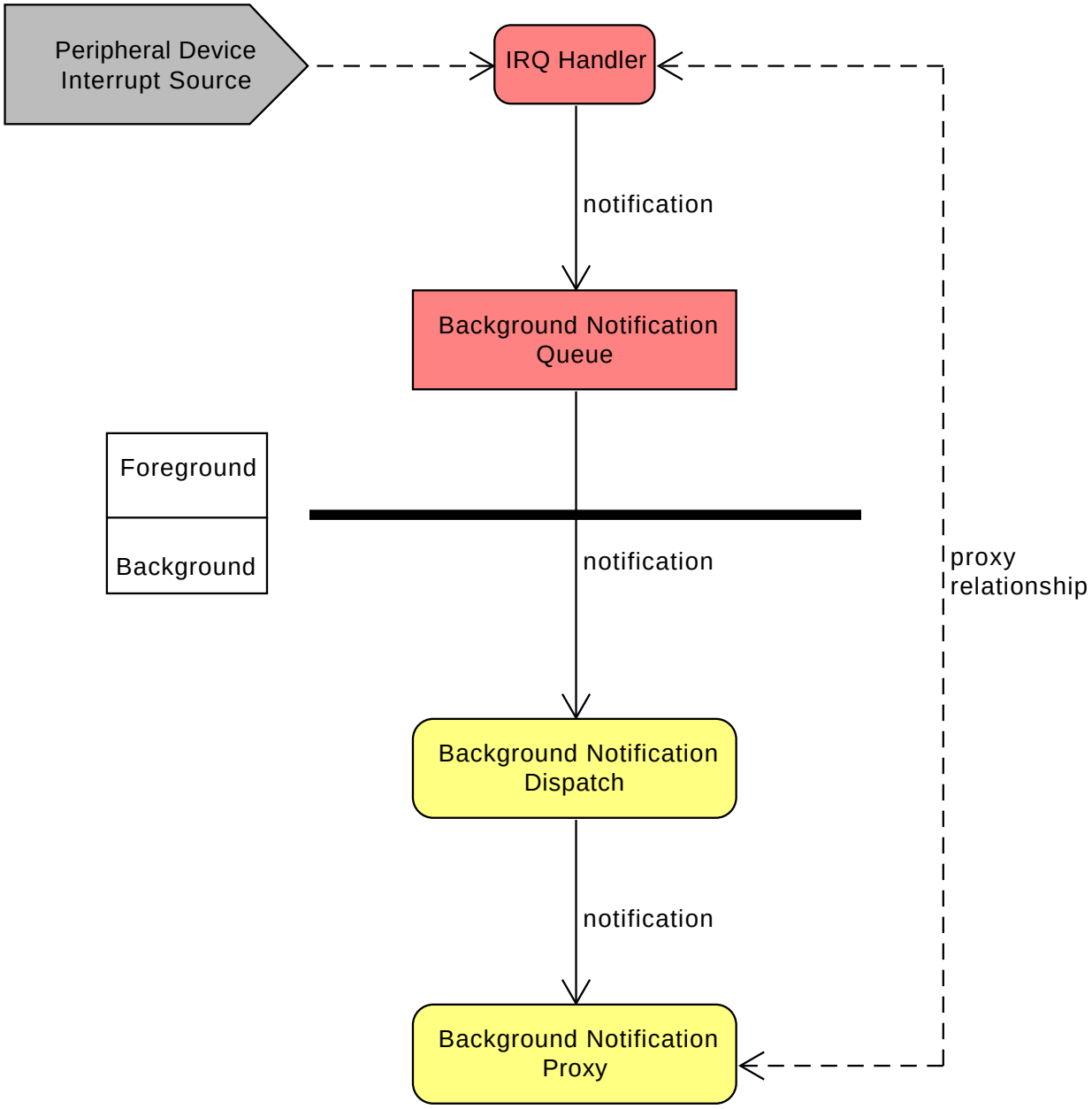

Figure 4.3: Background Notification Overview

This overview shows the relationship between an **IRQ Handler** and the **Background Notification Queue**. If an IRQ Handler needs to notify background processing of a happening in the environment, it inserts a notification into the **Background Notification Queue**. The **Background Notification Dispatch** code removes queued notifications and dispatches them. The dispatch results in executing a **Background Notification Proxy**. The IRQ handler and its notification information have a proxy relationship with the **Background Notification Handler**. The **Background Notification Proxy** can be viewed as a continuation function of the IRQ handler. The binding of a background notification proxy and a logical device is made at run time, *i.e.* device realm requests associated with devices that support some form of notification accept parameters to specify which background proxy function is to handle the notifications generated by the device. Note that there may be multiple **Background Notification Proxy** functions for any given logical device, depending upon the design of the operations of the peripheral device. The flexibility given by binding a logical device to a **Background Notification Proxy** at run time makes it much easier to run different

applications on an existing base of device realm code, *cf.* the binding of background requests to foreground proxies discussed in the last section where the binding is done at compile time. The compile-time binding of background requests to foreground proxy functions implies that the capabilities of a device are known when a system is built and the interfaces to those capabilities are, consequently, also fixed.

The previous figure is also a simplification of the foreground/background interaction. A mechanism is required for background processing to obtain the notifications for dispatch. As shown in the diagram, the **Background Notification Queue** is part of the foreground processing and thus requires privilege to access. The only way for **Background Notification Dispatch** to obtain a notification is to issue a system realm request to obtain one. The following diagram shows the details of using a system realm request to obtain a background notification placed in the **Background Notification Queue**.

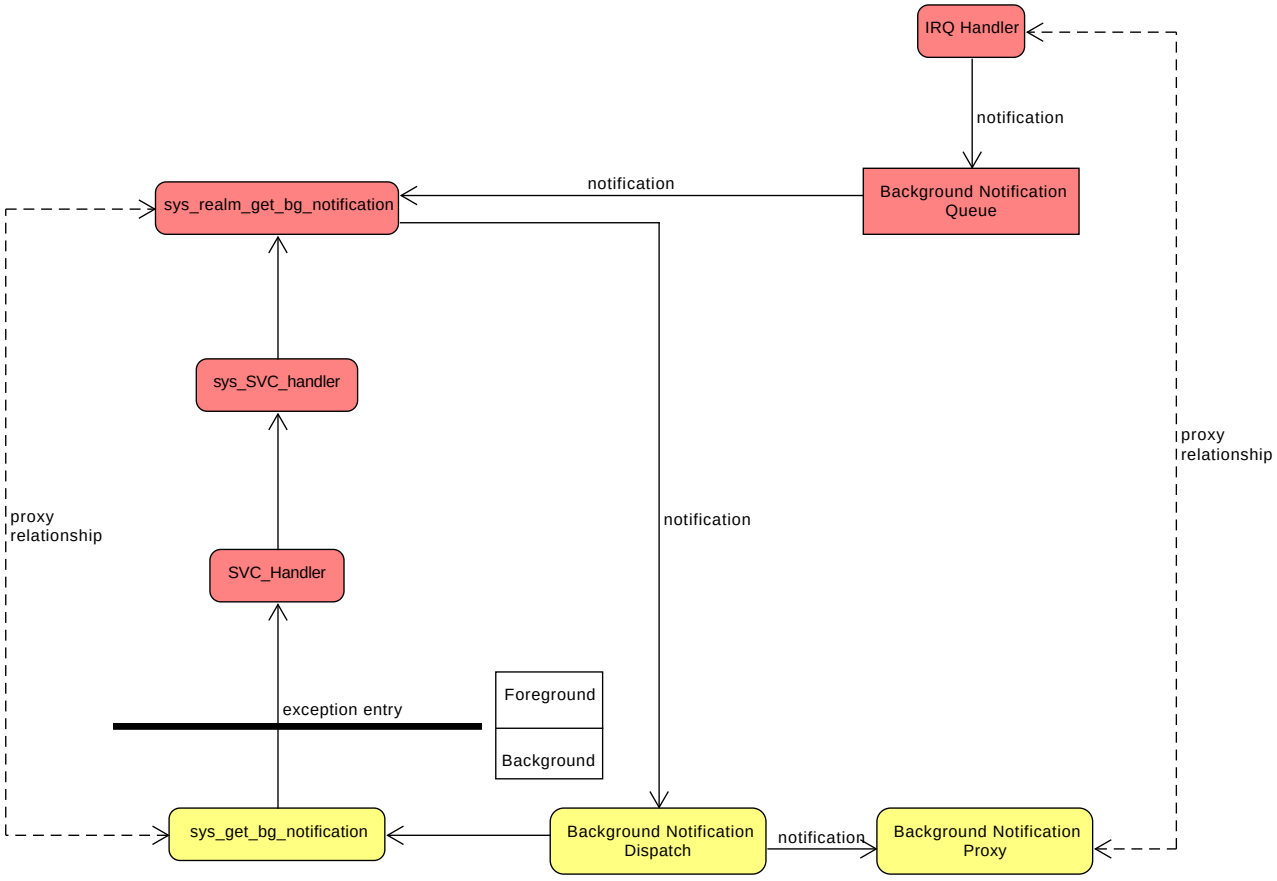

Figure 4.4: Background Notification Retrieval

In this figure, a system realm request is used to obtain a notification from the **Background Notification Queue**. The proxy relationship between the IRQ Handler and the **Background Notification Proxy** is established (at run time) prior to enabling any IRQ for the device (typically at device "open" time). When the **IRQ Handler** is run, it places a notification in the queue. **Background Notification Dispatch** then uses the `sys_get_bg_notification` function to request notifications from the queue. Notifications are *copied* from the foreground arena to the background arena by privileged execution. It is important that data flow is from privileged execution to unprivileged execution.

In the next sections, the design and code for the components of the foreground/background interface are shown. First, the `SVC` exception handler is considered. The discussion is focused on its major role as the conduit for the flow of control and data between the foreground and background arenas. Second, the details of the mechanism by which background requests reach their corresponding foreground proxy functions are shown. Third, the design of how IRQ handlers queue notifications to the background, including how the background notification queue operates, is shown.

SVC Interface

The SVC instruction is the only means provided by the processor architecture for unprivileged code to request privileged service[3]. This implies that the SVC handler must be able to pass input parameters and return results across the foreground/background boundary. Typically, exception handlers do not perform that function and it requires some intricate programming to happen. Specifically, the interface must:

- Pass data into the foreground service realms as arguments. The argument passing conventions of the AAPCS must be used. The AACPS, in its most general case, is a complex set of conventions. This design adheres to the basic rules which pass the first four word sized arguments to a called function in the R0 - R3 registers.
- Return data back to the background as the results of a request.
- Have the functions which provide the privileged service in the foreground be invoked as ordinary "C" functions using the parameter passing conventions of the AAPCS. Specifically, avoiding function interfaces which have, as an argument, a pointer back into a stack frame is a goal. It is undesirable to force the foreground proxy functions to be aware of and tied to the layout of processor resources such as stack frames. This is deemed an unnecessary cognitive burden for writing foreground proxy functions.
- Segment the foreground processing into the service realms previously discussed.
- Make all argument passing and returned results of the SVC handler interface agnostic to the data types of the values being passed. This makes the SVC handler strictly a control/data transportation mechanism with no knowledge of the meaning or use of the passed data values.

There are five data values at which can be used in the SVC interface design:

1. The value of the immediate operand of the SVC instruction itself. Part of the instruction encoding is an 8-bit value which can be chosen as necessary.
2. Four 32-bit values contained in the R0 - R3 registers.

The SVC interface design allocates the five data values as follows:

SVC immediate operand
: Selects between the system realm and the device realm. The system realm is encoded as 0 and the device realm as 1. No other immediate operand values are supported.

req ⇒ R0
: A request identifier. The interpretation of the req value is determined by the individual service realm handlers.

input ⇒ R1
: A pointer to the input arguments, if any, of the operation. Arguments to the foreground proxy function are marshaled into a memory object and passed by pointer reference.

output ⇒ R2
: A pointer to where the output of the operation, if any, is returned. The pointer is to a memory object where output results from the proxy function are placed.

error ⇒ R3
: A pointer to where auxiliary error information, if any, is returned. The pointer is to a memory object where additional information about the cause of an error may be placed.

[3] Of course, the reality is more complicated. There are means for application code to request an IRQ to be made pending. But even that requires system code to have initialized a register to allow it.

Only the `req` argument is required. The pointers may be passed as `NULL` if the proxy operation does not require additional arguments of that type. Argument values passed though the `SVC` exception have no type interpretation. Data typing is lost upon exception entry and must be regained by the invoked foreground proxy function.

The return value of the `SVC` interface is an integer. A return of zero indicates success. A negative valued return indicates an error. The encoding of the error returns is as follows (the return value being the unary negation of the following values).

```
<<svc req errors: constants>>=
#define ERR_SUCCESS             0               /* place holder */
#define ERR_UNKNOWN_REALM       1
#define ERR_UNKNOWN_DEVICE      2
#define ERR_UNKNOWN_REQUEST     3
#define ERR_UNKNOWN_INSTANCE    4
#define ERR_INVALID_PARAM       5
#define ERR_OPERATION_FAILED    6
```

ERR_UNKNOWN_REALM
The realm to which the request was directed is unknown, *i.e.* the request failed to be identified as either a system realm request or a device realm request.

ERR_UNKNOWN_DEVICE
The class of device to which the request was made does not exist.

ERR_UNKNOWN_REQUEST
The type of request is not known.

ERR_UNKNOWN_INSTANCE
The instance of the device to which the request was made does not exist.

ERR_INVALID_PARAM
There was an error in a passed parameter.

ERR_OPERATION_FAILED
The operation implied by the request failed. For some requests, the error return information contains more specific indications of the failure.

It is useful to map error numbers to strings for use in messages.

```
<<svc req errors: external function declarations>>=
extern char const *const svc_err_string(int error) ;
```

error
An SVC error number. The error number may be negative as it is returned from the various background request functions.

Returns a human readable string representing the error number.

```
<<svc req errors: external function definitions>>=
char const *const
svc_err_string(
    int error)
{
    static char const *const error_strings[] = {
        [ERR_SUCCESS] = "Success",
        [ERR_UNKNOWN_REALM] = "Unknown realm",
        [ERR_UNKNOWN_DEVICE] = "Unknown device",
        [ERR_UNKNOWN_REQUEST] = "Unknown request",
        [ERR_UNKNOWN_INSTANCE] = "Unknown instance",
```

```
        [ERR_INVALID_PARAM] = "Invalid parameter",
        [ERR_OPERATION_FAILED] = "Operation failed",
    } ;

    int index = abs(error) ;

    return index < COUNTOF(error_strings) ?
            error_strings[index] : "Unknown error number" ;
}
```

To accomplish the design goals for the SVC interface requires assembly language programming. The processing may be broken down as follows.

1. Determine the immediate operand value in the SVC instruction.

   To obtain the immediate operand in the instruction, the return address which was placed on the stack during exception entry is examined. This step is complicated by needing to determine which stack (PSP or MSP) was in use when the SVC instruction was executed.

2. Using the value from the SVC instruction, select the proper foreground *service realm* handler.

   A simple jump table is used to select the realm handler corresponding to the SVC instruction operand. It is also necessary to verify the value passed as the SVC operand to insure it doesn't exceed the bounds of the jump table.

3. Copy the four input arguments from the exception frame into registers in accordance with the AAPCS.

   Because of the *late arrival* and *tail chaining* optimizations of the processor, the values of scratch registers *cannot* be used directly. Late arriving or tail chained interrupts may change the values of the scratch registers from what they were when the SVC instruction was executed by the background processing. Since the AAPCS passes the first four arguments in the R0 - R3 scratch registers, their values must be obtained from the exception frame where they were placed when the SVC instruction executed and where they have not been disturbed in the case of late arrival or tail chaining of interrupts.

4. Invoke the appropriate realm service handler.

5. Upon return, copy the return value in the **R0** register back to the exception frame.

   Since the exception return mechanism of the core restores the registers values from the stack, any return value must be copied out of the R0 register onto the stack and then, from the exception frame on the stack, the values are loaded back into the registers by the processor as part of the exception return.

6. Return from the exception back to unprivileged mode.

The following code implements the required interface. Detailed commentary follows the code.

```
<<svc handler: external function definitions>>=
__attribute__((naked))
void
SVC_Handler(void)
{
    __asm__ volatile
    (
        "tst    lr,#4                           \n\t"
        "ite    eq                              \n\t"
        "mrseq  ip,msp                          \n\t"
        "mrsne  ip,psp                          \n\t"
        "ldr    r0,[ip, #24]                    \n\t"
        "ldrb   r0,[r0, #-2]    @ SVC immed operand \n\t"
        "cmp    r0,#(2f-1f)/4   @ number of realms  \n\t"
        "ittt   cs                              \n\t"
        "movcs  r0,#-1                          \n\t"
        "strcs  r0,[ip]                         \n\t"
        "bxcs   lr                              \n\t"
        "adr    r1,1f                           \n\t"
        "ldr    r1,[r1, r0, lsl #2]             \n\t"
```

```
21          "push   {r1}                            \n\t"
22          "ldm    ip,{r0-r3}  @ regs from exc frame   \n\t"
23          "pop    {ip}        @ handler pointer   \n\t"
24          "push   {lr}                            \n\t"
25          "blx    ip          @ handler call      \n\t"
26          "pop    {lr}                            \n\t"
27          "tst    lr,#4                           \n\t"
28          "ite    eq                              \n\t"
29          "mrseq  ip,msp                          \n\t"
30          "mrsne  ip,psp                          \n\t"
31          "str    r0,[ip]     @ return value      \n\t"
32          "bx    lr                               \n\t"
33          ".align 2                               \n"
34          "1:                                     \n\t"
35          ".word  sys_SVC_handler                 \n\t"
36          ".word  dev_SVC_handler                 \n"
37          "2:                                     \n"
38      ) ;
39  }
```

Line 2

The `naked` attribute requests the compiler *not* to include the usual function prologue and epilogue instructions. This compiler directive was used previously by the Default_Handler.

Lines 8 - 11

This is the usual idiom to determine which stack was in use when an exception was activated. In this case, the proper stack pointer value is placed into the `IP` (aka `R12`) register. This is also a scratch register, so it may be used without preserving. Normally, it is used for *veneers* which are generated at link time when the offset for branching is larger than can be held as an immediate operand in an instruction. Here it serves as a useful place to hold the pointer to the working stack.

Lines 12 - 13

These instructions place the immediate operand value from the `SVC` instruction into `R0`. The value of the stacked program counter is offset 24 bytes into the exception frame. By stepping back from the return address by 2 bytes, the `SVC` immediate operand is found. It is 1 byte long and conveniently located in a single byte by itself. The immediate operand value is used as an array index into a jump table of request handlers.

Line 14

Compute the number of foreground realm service handler by address arithmetic and use the result as an immediate operand. The difference between the end of the jump table and its beginning divided by the size of each entry is the count of entries in the jump table. The `2f` and `1f` notation refers to the labels at lines 37 and 34, respectively. By comparing the number of entries in the jump table to the `SVC` immediate operand, it is determined if the bounds of the jump table array are exceeded.

Lines 15 - 18

The `cmp` instruction sets the carry flag if the `SVC` operand is too large. Using conditional instructions, -1 is returned to indicate a bad operand. Note how the return value must be placed back into the exception frame at line 17.

Lines 19 - 20

Compute the address of the foreground realm handler function by indexing into the jump table. The table begins as the `1f:` label (line 34). The index in `R0` is scaled by 4 (`lsl #2`) to account for the number of bytes each pointer consumes, *i.e.* transform an element count into a byte count before fetching the handler function pointer from the jump table.

Line 21

At this point, `R0` and `R1` are not used further and `R1` holds a pointer to the realm handler function. Before invoking the realm handler, load in the scratch registers from the working stack. But doing that would overwrite `R1` where the handler function pointer has been placed. So it must be saved. Note it is always placed on the MSP since the processor is executing an exception handler.

Line 22

It takes only a single instruction to copy the stored values from the exception frame into the registers. Recall that `IP` contains the exception frame pointer value.

Line 23
Restore the value of the handler function in preparation to invoke it. The function pointer is placed in `IP` since it's value does not need to be preserved and `R0` - `R3` currently contain the function arguments.

Lines 24 - 25
The conventional idiom to invoke a function when the address is in a register. The `LR` must be preserved. At this point, `LR` contains the exception return value placed there when the `SVC` exception was entered.

Line 26
Restore the exception return value from the stack and place it back in `LR`. It is needed both as the return mechanism and to determine where the exception frame is located. Also, net stack usage is now zero.

Lines 27 - 31
Determine again where the exception frame is located by examining the exception return value and copy the `R0` register back into the exception frame. This is in preparation for the exception return which copies the values from the stack back into the registers. Every proxy function returns only a single integer. Other registers are purposely *not* copied back to the exception frame so as not to "leak" information to background processing.

Line 32
Returning from the exception causes the exception frame values from the stack to be restored to the registers.

Lines 34 - 37
The definition of the jump table for realm handlers.

Unprivileged Data Access

The `SVC` instruction provides a means of transferring *control* from unprivileged execution to privileged execution. The interface discussed in the last section allows for passing pointers to data structures which hold input, output, and error parameter data. The service realm handlers need access to that data and in this section transferring *data* between unprivileged and privileged execution is described.

A naive approach for transferring data would be to use directly the pointer passed to a foreground proxy function. But even casual examination shows that directly using pointers in privileged execution which were passed from unprivileged code is a recipe for undoing the attempts to isolate the two privilege realms.

Consider, for example, returning output from a background request. Let's say that the request was to read data from a device and return that data to the background at an address specified in the request. If the device data is written directly to an address passed in as an input parameter, then the memory access is done in privileged execution mode. Should the background processing make a mistake, either intentional or not, the device data could be written to memory used by privileged code. Critically, the naive approach allows the privileged code to become the unwitting patsy for unprivileged code to write into memory where it would otherwise not be allowed to write.

Fortunately, this issue is well understood. The essential rule is that any memory access by privileged code which transfers data to or from unprivileged code should be performed in an unprivileged manner. The ARMv7m architecture provides specific instructions to perform register loads and stores in an unprivileged manner. The `LDR??T` instructions allow reading byte, half-word, and word sized values from memory as either signed or unsigned integers and the access is performed in an unprivileged manner. Correspondingly, the `STR?T` instructions store byte, half-word, and word sized values into memory in an unprivileged manner. So the ability to perform unprivileged access from privileged execution is provided by the processor, but this means specific instructions must be used to do so. So, there is more systems programming and some assembly language to do.

Also note, that should an unprivileged function pass a pointer through the `SVC` interface to memory which requires privilege for access, using the unprivileged load and store instructions results is a Memory Management Fault or a Usage Fault.

Unprivileged memory access

The functions in this section provide access to the Cortex-M4 instruction that perform unprivileged access. Note that most of these are also available from the CMSIS header files. But CMSIS is missing the signed integer load instructions, *i.e.* `LDRSBT`, `LDRHBT`, and `LDRST` and the typing of the return values is also not quite what is desirable. So, it is necessary to roll up our sleeves and crank out some inline functions. There is a read and write instruction for each of the usual integer types. Note in

places, the same assembly instruction is used for different “C” argument types. For example, the processor makes no differences between storing a signed or unsigned value into memory because raw memory does not have a type. To eliminate compiler warnings, it is necessary to have separate functions for signed and unsigned quantities even if the resulting machine instruction is the same. The result is many lines of “C” code which, when optimized by the compiler, will result in a single machine instruction.

```
<<svc proxy: static inline functions>>=
static inline uint32_t
read_u8_unpriv(
    uint8_t const *src)
{
    uint32_t result ;

    __asm__ volatile (
        "ldrbt %[result],%[addr]"
        : [result] "=r" (result)
        : [addr] "Q" (*src)
    ) ;

    return result ;
}
```

```
<<svc proxy: static inline functions>>=
static inline void
write_u8_unpriv(
    uint8_t value,
    uint8_t *dst)
{
    __asm__ volatile (
        "strbt %[value],%[addr]"
        :
        : [addr] "Q" (*dst),
          [value] "r" (value)
    ) ;
}
```

```
<<svc proxy: static inline functions>>=
static inline int32_t
read_i8_unpriv(
    int8_t const *src)
{
    int32_t result ;

    __asm__ volatile (
        "ldrsbt %[result],%[addr]"
        : [result] "=r" (result)
        : [addr] "Q" (*src)
    ) ;

    return result ;
}
```

```
<<svc proxy: static inline functions>>=
static inline void
write_i8_unpriv(
    int8_t value,
    int8_t *dst)
{
    __asm__ volatile (
        "strbt %[value],%[addr]"
        :
        : [addr] "Q" (*dst),
```

```
          [value] "r" (value)
    ) ;
}
```

```
<<svc proxy: static inline functions>>=
static inline uint32_t
read_u16_unpriv(
    uint16_t const *src)
{
    uint32_t result ;

    __asm__ volatile (
        "ldrht %[result],%[addr]" :
        [result] "=r" (result) :
        [addr] "Q" (*src)
    ) ;

    return result ;
}
```

```
<<svc proxy: static inline functions>>=
static inline void
write_u16_unpriv(
    uint16_t value,
    uint16_t *dst)
{
    __asm__ volatile (
        "strht %[value],%[addr]"
        :
        : [addr] "Q" (*dst),
          [value] "r" (value)
    ) ;
}
```

```
<<svc proxy: static inline functions>>=
static inline int32_t
read_i16_unpriv(
    int16_t const *src)
{
    int32_t result ;

    __asm__ volatile (
        "ldrsht %[result],%[addr]" :
        [result] "=r" (result) :
        [addr] "Q" (*src)
    ) ;

    return result ;
}
```

```
<<svc proxy: static inline functions>>=
static inline void
write_i16_unpriv(
    int16_t value,
    int16_t *dst)
{
    __asm__ volatile (
        "strht %[value],%[addr]"
        :
        : [addr] "Q" (*dst),
```

```
          [value] "r" (value)
    ) ;
}
```

```
<<svc proxy: static inline functions>>=
static inline unsigned
read_unsigned_unpriv(
    unsigned const *src)
{
    unsigned result ;

    __asm__ volatile (
        "ldrt %[result],%[addr]" :
        [result] "=r" (result) :
        [addr] "Q" (*src)
    ) ;

    return result ;
}
```

```
<<svc proxy: static inline functions>>=
static inline void
write_unsigned_unpriv(
    unsigned value,
    unsigned *dst)
{
    __asm__ volatile (
        "strt %[value],%[addr]"
        :
        : [addr] "Q" (*dst),
          [value] "r" (value)
    ) ;
}
```

```
<<svc proxy: static inline functions>>=
static inline int
read_int_unpriv(
    int const *src)
{
    int result ;

    __asm__ volatile (
        "ldrt %[result],%[addr]" :
        [result] "=r" (result) :
        [addr] "Q" (*src)
    ) ;

    return result ;
}
```

```
<<svc proxy: static inline functions>>=
static inline void
write_int_unpriv(
    int value,
    int *dst)
{
    __asm__ volatile (
        "strt %[value],%[addr]"
        :
        : [addr] "Q" (*dst),
```

```
          [value] "r" (value)
    ) ;
}
```

```
<<svc proxy: static inline functions>>=
static inline uint32_t
read_u32_unpriv(
    uint32_t const *src)
{
    uint32_t result ;

    __asm__ volatile (
        "ldrt %[result],%[addr]" :
        [result] "=r" (result) :
        [addr] "Q" (*src)
    ) ;

    return result ;
}
```

```
<<svc proxy: static inline functions>>=
static inline void
write_u32_unpriv(
    uint32_t value,
    uint32_t *dst)
{
    __asm__ volatile (
        "strt %[value],%[addr]"
        :
        : [addr] "Q" (*dst),
          [value] "r" (value)
    ) ;
}
```

```
<<svc proxy: static inline functions>>=
static inline int32_t
read_i32_unpriv(
    int32_t const *src)
{
    int32_t result ;

    __asm__ volatile (
        "ldrt %[result],%[addr]" :
        [result] "=r" (result) :
        [addr] "Q" (*src)
    ) ;

    return result ;
}
```

```
<<svc proxy: static inline functions>>=
static inline void
write_i32_unpriv(
    int32_t value,
    int32_t *dst)
{
    __asm__ volatile (
        "strt %[value],%[addr]"
        :
        : [addr] "Q" (*dst),
```

```
          [value] "r" (value)
    ) ;
}
```

To make it easier to select which function to use, a macro using the `_Generic()` selector that is available in `C11` is used. This language construct allows you to implement a form of function overloading. The compiler selects the correct unprivileged operation based on the data type of the data source or destination.

```
<<svc proxy: macros>>=
#define READ_UNPRIV(src)   _Generic((src),          \
        uint8_t *: read_u8_unpriv,                  \
        uint8_t const *: read_u8_unpriv,            \
        int8_t *: read_i8_unpriv,                   \
        int8_t const *: read_i8_unpriv,             \
        uint16_t *: read_u16_unpriv,                \
        uint16_t const *: read_u16_unpriv,          \
        int16_t *: read_i16_unpriv,                 \
        int16_t const *: read_i16_unpriv,           \
        uint32_t *: read_u32_unpriv,                \
        uint32_t const *: read_u32_unpriv,          \
        int32_t *: read_i32_unpriv,                 \
        int32_t const *: read_i32_unpriv,           \
        unsigned *: read_unsigned_unpriv,           \
        unsigned const *: read_unsigned_unpriv,     \
        int *: read_int_unpriv,                     \
        int const *: read_int_unpriv                \
    )(src)
```

```
<<svc proxy: macros>>=
#define WRITE_UNPRIV(v, dst)   _Generic((dst),      \
        uint8_t *: write_u8_unpriv,                 \
        int8_t *: write_i8_unpriv,                  \
        uint16_t *: write_u16_unpriv,               \
        int16_t *: write_i16_unpriv,                \
        uint32_t *: write_u32_unpriv,               \
        int32_t *: write_i32_unpriv,                \
        unsigned *: write_unsigned_unpriv,          \
        int *: write_int_unpriv                     \
    )(v, dst)
```

With all the inline assembly and generic choice macros in place, you can write versions of `memcpy` to copy to and from unprivileged memory. Note these function take arguments in the same order as the standard library `memcpy()` function, *i.e.* destination then source. However, which pointer refers to unprivileged memory, of course, switches depending upon the copy direction.

```
<<svc proxy: static inline functions>>=
static inline void
memcpy_from_unpriv(
    void *dest,
    void const *unpriv_src,
    size_t n)
{
    uint8_t *d = dest ;
    uint8_t const *s = unpriv_src ;

    while (n-- != 0) {
        *d++ = READ_UNPRIV(s) ;
        s++ ;
    }
}
```

```
<<svc proxy: static inline functions>>=
static inline void
memcpy_to_unpriv(
    void *unpriv_dest,
    void const *src,
    size_t n)
{
    uint8_t *d = unpriv_dest ;
    uint8_t const *s = src ;

    while (n-- != 0) {
        uint8_t byte = *s++ ;
        WRITE_UNPRIV(byte, d) ;
        d++ ;
    }
}
```

Service Parameters

There is common code that can be factored out to obtain a consistent interface for the `input`, `output`, and `error` parameters that form the arguments to `SVC` requests. Each of the foreground proxy functions has to handle passed-in arguments. But each argument is a different structure for each request. Separate mechanisms are factor the common code from the specific to provide mechanisms for common operations.

The only common aspect of the parameter arguments is the field giving the size of the parameter structure.

```
<<svc param: data type declarations>>=
typedef struct {
    size_t block_size ;
} SVC_RequestParam ;
```

block_size
: The number of bytes occupied by the request parameter.

Each system or device realm operation is expected to supply specific type definitions for each parameter the operation accepts. Those specific definitions define structures holding the parameter values and each structure must have a member of `SVC_RequestParam` type as its first member. This will allow us to *down cast* in the common handling code. To help in those definitions, a pre-processor macro is used.

```
<<svc param: macros>>=
#define DECLARE_REQUEST_PARAM(type, members)    \
typedef struct {                                \
    SVC_RequestParam ;                          \
    members                                     \
} type
```

Important
Because the use of unnamed structures is common in the service parameters, an extension to GCC is used. The extension is invoked using the `-fplan9-extensions`. The description of the effect of this extension is:

> As permitted by ISO C11 and for compatibility with other compilers, GCC allows you to define a structure or union that contains, as fields, structures and unions without names. For example:
>
> ```
> struct {
>   int a;
>   union {
>     int b;
>     float c;
>   };
>   int d;
> } foo;
> ```
>
> In this example, you are able to access members of the unnamed union with code like `foo.b`. Note that only unnamed structs and unions are allowed, you may not have, for example, an unnamed `int`.
>
> You must never create such structures that cause ambiguous field definitions. For example, in this structure:
>
> ```
> struct {
>   int a;
>   struct {
>     int a;
>   };
> } foo;
> ```
>
> 
>
> it is ambiguous which `a` is being referred to with `foo.a`. The compiler gives errors for such constructs.
>
> Unless `-fms-extensions` is used, the unnamed field must be a structure or union definition without a tag (for example, `struct { int a; };`). If `-fms-extensions` is used, the field may also be a definition with a tag such as `struct foo { int a; };`, a reference to a previously defined structure or union such as `struct foo;`, or a reference to a `typedef` name for a previously defined structure or union type.
>
> The option `-fplan9-extensions` enables `-fms-extensions` as well as two other extensions. First, a pointer to a structure is automatically converted to a pointer to an anonymous field for assignments and function calls. For example:
>
> ```
> struct s1 { int a; };
> struct s2 { struct s1; };
> extern void f1 (struct s1 *);
> void f2 (struct s2 *p) { f1 (p); }
> ```
>
> In the call to `f1` inside `f2`, the pointer `p` is converted into a pointer to the anonymous field.
>
> Second, when the type of an anonymous field is a `typedef` for a `struct` or `union`, code may refer to the field using the name of the `typedef`.
>
> ```
> typedef struct { int a; } s1;
> struct s2 { s1; };
> s1 f1 (struct s2 *p) { return p->s1; }
> ```
>
> These usages are only permitted when they are not ambiguous.
>
> — Section 6.63 *GCC manual*

```
<<svc proxy: external function declarations>>=
extern int
copy_in_svc_param(
    SVC_RequestParam const *const unpriv,
    size_t param_size,
    void *const param) ;
```

unpriv
A pointer to an unprivileged memory object containing the service request parameter.

param_size
The size in bytes of the destination object to receive the copy, *i.e.* the number of bytes pointed to by `param`.

param
A privileged memory pointer to the object which receives the copy of the data pointed to by `unpriv`.

The `copy_in_svc_param` function copies a service request parameters from `unpriv` to the memory pointed to by `param` which is `param_size` bytes long. The copy is performed in an unprivileged manner. The `param_size` argument must match the `block_size` field of `unpriv`.

The return value is 0, upon success and negative otherwise.

Implementation

```
<<svc proxy: external function definitions>>=
int
copy_in_svc_param(
    SVC_RequestParam const *const unpriv,
    size_t param_size,
    void *const param)
{
    rtcheck_return(unpriv != NULL && param_size != 0 && param != NULL,
            -ERR_INVALID_PARAM) ;                                    // ❶

    size_t block_size = READ_UNPRIV(&unpriv->block_size) ;
    rtcheck_return(block_size == param_size, -ERR_INVALID_PARAM) ;

    memcpy_from_unpriv(param, unpriv, param_size) ;
    return 0 ;
}
```

❶ The various flavors of the `rtcheck` functions all perform run time checks and tests. These functions are defined in the next chapter.

```
<<svc proxy: external function declarations>>=
extern int
copy_out_svc_result(
    SVC_RequestParam *const unpriv,
    size_t result_size,
    void const *const result) ;
```

unpriv
A pointer to an unprivileged memory object where the result is copied.

result_size
The size in bytes of the destination object to receive the copy, *i.e.* the number of bytes pointed to by `result`.

result
A privileged memory pointer to the object which is the source of data to be copied.

The `copy_out_svc_param` function copies a service request result from privileged memory pointed to by `result`, which is `param_size` bytes long, to unprivileged memory memory pointed to by `unpriv`. The copy is performed in an unprivileged manner. The `block_size` field of `unpriv` must match the valueof `result_size`.

The return value is 0, upon success and negative otherwise.

Implementation

```
<<svc proxy: external function definitions>>=
int
copy_out_svc_result(
    SVC_RequestParam *const unpriv,
    size_t result_size,
    void const *const result)
{
    rtcheck_return(unpriv != NULL && result_size != 0 && result != NULL,
            -ERR_INVALID_PARAM) ;

    size_t block_size = READ_UNPRIV(&unpriv->block_size) ;
    rtcheck_return(block_size == result_size, -ERR_INVALID_PARAM) ;

    memcpy_to_unpriv(unpriv, result, result_size) ;
    return 0 ;
}
```

System Realm Request Interface

This section shows the details of how a background request function for the system realm uses the `SVC` interface to transfer control to its foreground proxy function. The interface expects each system realm request available to background processing to marshal its inputs into a value whose type is a structure where the first member corresponds to the `SVC_RequestParam` type. This determines the total number of bytes of the parameter.

For system realm requests, the `req` values are unsigned consecutive integers starting at zero. A data type is defined for the `req` argument.

```
<<sys realm param: data type declarations>>=
typedef uint32_t SVC_SysRequest ;
```

Making System Realm Requests

The purpose of the `sys_realm_svc_call` function is to coax the compiler into loading the operands into the proper registers and then to execute the `SVC` instruction with the appropriate immediate operand. This results in the processor registers containing the correct values to match the expectations of the `SVC` handler.

```
<<svc req: external declarations>>=
extern int
sys_realm_svc_call(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

Implementation

```
<<svc req: external function definitions>>=
int
sys_realm_svc_call(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    register int result __asm__ ("r0") ;

    __asm__ volatile
    (
        "svc    #0" : "=r" (result) : : "cc", "memory"
    ) ;

    return result ;
}
```

Handling System Realm Requests

As was shown in the description of the SVC handler, finding the appropriate system realm foreground proxy function first goes through a system realm handler. The system realm handler uses the `req` argument to locate the foreground proxy function. Note the interface to the system realm handler is the same as that of the `sys_realm_svc_call` function. This is to be expected since the `SVC` handler itself simply passes through the value of the scratch registers `R0` - `R3`.

The implementation of the system realm handler uses a jump table. The jump table is constructed in an initialized array at compile time. To build the jump table, it is less verbose to create a `typedef` for the function pointers.

```
<<sys realm param: data type declarations>>=
typedef int (*SVC_SysRequestProxy)(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<svc handler: static function definitions>>=
__attribute__((used))                                    // ❶
static int
sys_SVC_handler(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
```

```
    static SVC_SysRequestProxy const sys_req_proxies[] = {
        <<svc entry: system request functions>>
    } ;

    rtcheck_max_return(req, COUNTOF(sys_req_proxies), -ERR_UNKNOWN_REQUEST) ;

    SVC_SysRequestProxy const proxy = sys_req_proxies[req] ;
    assert(proxy != NULL) ;

    return proxy != NULL ? proxy(req, input, output, error) : -ERR_UNKNOWN_REQUEST ;
}
```

❶ Since the function is referenced by inline assembly, and since the compiler does *not* interpret the contents of inline assembly, the compiler must be informed that this function is actually used despite it not being invoked from any "C" function in the file.

Abnormal Termination Background Request Function

To demonstrate the details of the system realm interface, a simple background request for an abnormal termination is shown. Previously, the function, SystemAbend(), was defined. Because `SystemAbend` accesses privileged registers, it may not be invoked from unprivileged code without causing a system fault. For background processing to terminate the system, the `sys_abend()` function is defined and associated to a proxy in the foreground that executes `SystemAbend`.

The following steps establish a pattern for creating system realm requests. First, a function interface for the background request must be defined. The `sys_abend()` function is particularly simple since it takes no arguments and, ultimately, never returns.

```
<<sys svc req: external declarations>>=
extern noreturn void
sys_abend(void) ;
```

The `sys_abend` function forces an abnormal termination of the system. Ultimately, it executes the `SystemAbend()` function. It is not expected to return and by default forces a system reset. However, `SystemAbend` is defined as a *weak* symbol and can be overridden to obtain application specific behavior.

It has a simple implementation in the background since no arguments are passed.

Implementation

```
<<sys svc req: external definitions>>=
noreturn void
sys_abend(void)
{
    (void)sys_realm_svc_call(SYS_ABEND_REQ, NULL, NULL, NULL) ;      // ❶
    abort() ;
}
```

❶ By itself, this line causes the compiler to emit a "noreturn function does return" warning. There does not seem to be any way to for the compiler to ignore it. The foreground proxy function, in this case, does not return even if `sys_realm_svc_call` does return in every other case. The solution is to invoke a function with a `noreturn` declaration, even though it is never expected to execute.

The request is encoded as an integer to identify it.

```
<<sys realm param: constants>>=
#define SYS_ABEND_REQ       0
```

The foreground proxy function is added to the system realm dispatch jump table.

```
<<svc entry: system request functions>>=
[SYS_ABEND_REQ] = sys_realm_abend,
```

Abnormal Termination Foreground Proxy Function

The other half of the system realm request is implemented by the foreground proxy function for `sys_abend`.

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_abend(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

The implementation simply invokes `SystemAbend` which is already provided.

Implementation

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_abend(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(req == SYS_ABEND_REQ) ; (void)req ;
    assert(input == NULL) ; (void)input ;
    assert(output == NULL) ; (void)output;
    assert(error == NULL) ; (void)error ;

    SystemAbend() ;
    return 0 ;                                          // ❶
}
```

❶ This function does not return, but the compiler needs to see a return value which matches the function prototype.

Device Realm Request Interface

The last section showed the details of how a system realm request for abnormal termination is implemented by a foreground proxy. In this section, background requests targeted at the device realm are described. The peripheral devices play a central role in the flow of data and control in the system and this section lays out the pattern of device realm request processing.

In the system realm, the `req` argument was an integer encoded to identify a particular operation. There is a one-to-one mapping between the `req` value and a function to fulfill the request. In the device realm, the `req` argument is interpreted differently. For devices, the mapping of the request to an operation function requires more information, namely:

- The *class* of the device. This is a numerical encoding that identifies the general category of the device.
- The *operation* on the device. Each device class provides it's own set of operations which are specific to the device. There is no attempt to use any type of polymorphic interface for all devices. Some devices may follow more traditional "file" oriented behavior where *open*, *close*, *read*, and *write* might be appropriate. But for many microcontroller devices, modelling them as streams of bytes does not necessarily give a convenient interface for device interaction. Many devices are simply control oriented and do not transfer data. Such devices stretch the file metaphor beyond recognition.

- The *instance* of the device. Many devices have multiple, independent instances of hardware which can be operated upon separately. All instances of a device support the same operations.

In this context, the device class may refer to a particular peripheral device *or* it may be considered a *logical* device which is some combination of physical peripherals. For example, accurately timed sampling of an *Analog to Digital Converter* (ADC), can involve a timer peripheral to trigger the signal conversion. This insures that the conversion rate is stable and contains less jitter. It may also involve a *Direct Memory Access* (DMA) controller to move the data off the ADC and into memory. However, from a software context, it is preferable to treat such an arrangement of cooperating peripherals as a single "device." Sometimes, a peripheral device may even be split into multiple logical devices. These decompositions are an element of the design for how device realm processing interacts with the SOC hardware.

The device realm must also account for there being multiple *instances* of a device. It is not uncommon to have multiple UART's or SPI buses on an SOC. It is convenient to view that situation as multiple instances of a particular device class. This view implies that all operations supported by the device class may be applied to any instance.

The background requests into the device realm must supply the information discussed above. Since there is a single 32-bit integer value as the argument, the class, operation, and instance are encoded as bit fields within the `req` argument.

1. The class of the device is held in bits 0 - 7. This is a sequential integer encoding starting at zero.
2. The requested operation is held in bits 8 - 15. Again, a simple sequential integer encoding is used to identify an operation. The set of operations on a particular class of device is fixed at compile time.
3. The instance of the device to which the operation applies is held in bits 16 - 23. Yet again, sequential integers which start at zero are used to identify the instance.
4. Bits 24 - 31 are unused and reserved for those cases where 8 bits may not be sufficient for encoding one of the three fields which make up the device `req` argument.

The choice of the encoding for device realm entities is intended to make the identifiers useful as an array index and the encoded values are used directly in the implementation as indices.

To make the encoding of device class, operation, and instance more convenient in the code, a set of types and functions are defined to encapsulate the specifics of the encoding rules.

```
<<dev realm param: data type declarations>>=
typedef uint32_t SVC_DevRequest ;
```

```
<<dev svc req: data type declarations>>=
typedef uint8_t SVC_DevClass ;
typedef uint8_t SVC_DevInstance ;
```

```
<<dev realm param: data type declarations>>=
typedef uint8_t SVC_DevOperation ;
```

```
<<dev realm param: constants>>=
#define DEV_REQ_CLASS_WIDTH         8
#define DEV_REQ_CLASS_OFFSET        0

#define DEV_REQ_OPERATION_WIDTH     8
#define DEV_REQ_OPERATION_OFFSET    (DEV_REQ_CLASS_OFFSET + DEV_REQ_CLASS_WIDTH)

#define DEV_REQ_INSTANCE_WIDTH      8
#define DEV_REQ_INSTANCE_OFFSET     (DEV_REQ_OPERATION_OFFSET + DEV_REQ_OPERATION_WIDTH)
```

Encoding a class/operation/instance triple into a device request parameter is a simple bit packing operation.

```
<<dev realm param: static inline definitions>>=
static inline SVC_DevRequest
dev_req_encode(
    SVC_DevClass class,
```

```
    SVC_DevOperation operation,
    SVC_DevInstance instance)
{
    SVC_DevRequest req = 0 ;
    req = btwd_bits_insert(req, class, DEV_REQ_CLASS_WIDTH, DEV_REQ_CLASS_OFFSET) ;
    req = btwd_bits_insert(req, operation, DEV_REQ_OPERATION_WIDTH,
            DEV_REQ_OPERATION_OFFSET) ;
    req = btwd_bits_insert(req, instance, DEV_REQ_INSTANCE_WIDTH,
            DEV_REQ_INSTANCE_OFFSET) ;
    return req ;
}
```

Three decoding operations are provided to obtain the component parts of the device realm request parameter.

```
<<dev realm proxy: static inline definitions>>=
static inline SVC_DevClass
dev_req_extract_class(
    SVC_DevRequest req)
{
    return btwd_bits_extract(req, DEV_REQ_CLASS_WIDTH, DEV_REQ_CLASS_OFFSET) ;
}
```

```
<<dev realm proxy: static inline definitions>>=
static inline SVC_DevOperation
dev_req_extract_operation(
    SVC_DevRequest req)
{
    return btwd_bits_extract(req, DEV_REQ_OPERATION_WIDTH, DEV_REQ_OPERATION_OFFSET) ;
}
```

```
<<dev realm proxy: static inline definitions>>=
static inline SVC_DevInstance
dev_req_extract_instance(
    SVC_DevRequest req)
{
    return btwd_bits_extract(req, DEV_REQ_INSTANCE_WIDTH, DEV_REQ_INSTANCE_OFFSET) ;
}
```

Since requests are handle in a privileged environment, input validation must be done to catch any incorrectly supplied values. The validation of request values is particularly important since the request field values are used as array indices.

```
<<dev realm proxy: static inline definitions>>=
static inline int
dev_req_validate(
    SVC_DevRequest req,
    int op_count,
    int inst_count)
{
    uint32_t op = dev_req_extract_operation(req) ;
    rtcheck_max_return(op, op_count, -ERR_UNKNOWN_REQUEST) ;

    uint32_t inst = dev_req_extract_instance(req) ;
    rtcheck_max_return(inst, inst_count, -ERR_UNKNOWN_INSTANCE) ;

    return 0  ;
}
```

Making Device Realm Requests

Just as for the system realm, a function to execute the `SVC` instruction with the arguments in the appropriate register and with the `SVC` immediate operand set to 1 is defined.

```
<<svc req: external declarations>>=
extern int
dev_realm_svc_call(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

Implementation

```
<<svc req: external function definitions>>=
int
dev_realm_svc_call(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    register int result __asm__ ("r0") ;

    __asm__ volatile
    (
        "svc    #1"  : "=r" (result) : : "cc", "memory"
    ) ;

    return result ;
}
```

Handling Device Realm Requests

The device realm, like the system realm, has a handler function which is selected and invoked directly by the `SVC` handler.

The implementation follows the same pattern as `sys_SVC_handler`. In this case, the first level of decomposition is based on the class of the device and the jump table selects the handler function based on device class.

As usual, function pointers with their arguments are easier to handle with a `typedef`.

```
<<dev realm param: data type declarations>>=
typedef int (*SVC_DevRequestProxy)(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<svc handler: static function definitions>>=
__attribute__((used))
static int
dev_SVC_handler(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    static SVC_DevRequestProxy const dev_req_classes[] = {
        <<svc entry: device request classes>>
    } ;

    SVC_DevClass dev_class = dev_req_extract_class(req) ;              // ❶
    rtcheck_max_return(dev_class, COUNTOF(dev_req_classes),
            -ERR_UNKNOWN_DEVICE) ;
```

```
    SVC_DevRequestProxy const class_handler = dev_req_classes[dev_class] ;
    rtcheck_not_NULL_return(class_handler, -ERR_UNKNOWN_DEVICE) ;       // ❷

    return class_handler(req, input, output, error) ;
}
```

❶ The SVC handler for the device realm must validate the class number for the request as this is used as an index into a jump table.

❷ A device class handler must also be supplied when the system was built. This check insures that the pointer to the device class handler was actually placed in the jump table. Since it is a two step process to define a device class and to define the class operations handler, each part must be checked.

Background Notification Interface

The last section presented the interface used by background processing to request device realm services. The third major component of the foreground/background interface is the background notification queue.

As shown in the overview figure, a **Background Notification Queue** is the mechanism provided to IRQ handlers to make requests to a corresponding background notification proxy. The notification consist of:

- The device class value.
- The device instance value.
- A pointer to a background notification proxy function which acts upon the notification.
- A caller supplied *closure* value which is returned as part of the notification.

In code, the notification proxy is typed as:

```
<<dev svc req: data type declarations>>=
struct svc_dev_notification ;
typedef void (*SVC_DevNotifyProxy)(struct svc_dev_notification const *const params) ;
```

The closure value is large enough to hold a pointer. Note that the lifetime of the memory associated with any pointer must be managed by the background. The notification mechanism simply passes along the value which it was originally given. Background processing which sets up the binding to the notification proxy must make sure any pointer values are still valid by the time the notification is issued. That's a lot of words to say, "don't use pointers to automatic variables as notification closures."

```
<<dev svc req: data type declarations>>=
typedef uintptr_t SVC_DevNotifyClosure ;
```

The following is the generic base type for a notification. Device specific notifications are derived from the generic notification structure for those cases where additional information is supplied by the device.

```
<<dev svc req: data type declarations>>=
typedef struct svc_dev_notification {
    size_t block_size ;
    SVC_DevNotifyProxy notify_proxy ;
    SVC_DevNotifyClosure notify_closure ;
    SVC_DevClass device_class ;
    SVC_DevInstance device_instance ;
} SVC_DevNotification ;
```

Just as for declaring specific request parameter types, a macro saves some typing when declaring specific notification types where additional members are required.

```
<<dev svc req: data type declarations>>=
#define DECLARE_DEV_NOTIFICATION(type, members) \
typedef struct {                                \
    SVC_DevNotification ;                       \
    members                                     \
} type
```

By analogy to background requests, a peripheral device that places a notification into the queue has one or more proxy functions that are run in the background to handle the request. The background notification proxy can be considered as a continuation of the IRQ handler which computes the system response to the environmental condition detected by the IRQ. The peripheral device code and proxy function must agree to the structure of the notification since typing is lost during the transit through the queue and must recovered by the background proxy function. It is an element of the design of the peripheral device control code as to which notifications it offers. Any additional data beyond that provided by the base notification structure must also be specified for both sides.

The remaining consideration of the notification design is to specify the time when the notification proxy function is bound to a notification offered by a device. For background *requests*, the binding between the background function which issues requests and the foreground proxy function which services them is made at compile time. This is the simplest approach and, given the characteristics of foreground requests, additional flexibility is not required. For background *notifications*, the proxy function is bound at run time. This implies that at least part of the configuration of a peripheral device is to specify the notification proxy functions to be executed in the background when the devices detects an actionable condition. Devices may offer to issue multiple types of notifications, depending upon the specific semantic actions associated with the device.

The advantages of using run time binding of the background callback are:

- Although the system is expected to run a single program at a time, during development many test programs and integration programs are run. If the notification proxy functions were bound at compile time, all application would be required to supply an implementation of the functions which complicates the build/test process.
- In some cases, the condition detected by the device control code is not interesting to the application and it would be more convenient *not* to require specifying a notification proxy function for an unused condition, *i.e.* to specify a `NULL` function pointer. This knowledge may allow the peripheral device code to mask certain interrupt sources or take other actions to reduce its interaction with the environment.

Run time binding does imply that the peripheral device code must store the binding somewhere. This is usually of little consequence since device code must also store many other parameters used for controlling the peripheral itself.

It may seem dangerous to some readers to pass around “raw” function pointers between foreground and background. Certainly passing an incorrect function pointer can result in uncontrolled execution. It is not strictly necessary to use a function pointer. The background notifications could be encoded as integers (as they are for background requests). This requires keeping a mapping from the integer encoding to the corresponding proxy function which is used during dispatching the background notification. Such an encoding would also have to be constructed at run time and supplying an incorrect encoded integer when building the mapping would result in the same uncontrolled execution. In practice, the use of function pointers is simpler and does not cause significant problems.

Using function pointers does not jeopardize the separation between privileged and unprivileged execution. The notification proxy function is *not* executed by privileged code. Invocation of notification proxy functions is performed by background processing in unprivileged mode. Both the notification proxy function pointer and the notification closure data value traverse through the device control code to the background notification queue untouched.

Background Notification Queue Operations

The background notification queue is implemented as a bipartite buffer. Bipartite buffers have the desirable characteristic that they are implemented as a circular queue within an array, but an allocated slot always consists of contiguous memory locations. The particular bipartite buffer implementation used here has the following characteristics:

- The queue is in FIFO order.

- Queue requests are packet oriented (as opposed to a stream of bytes) and of arbitrary length (subject to the overall size of the buffer).

The referenced implementation defines a pre-processor macro to help in defining a bip-buffer. Like other system resources, the size of the buffer is set at compile time. Sizing the background notification buffer involves two considerations:

1. The size of a background notification and in particular the amount of data sent in the notification itself.
2. The number of notifications which may be queued during peek interrupt activity.

Both considerations are difficult to estimate with no prior knowledge of the application characteristics. Experience shows most background notifications are small in size, typically 16 bytes (the default notification size). Passing large amounts of data through the queue is not advisable, especially since there are other techniques for passing data and any data passed through the queue must be copied into the background. The worst case scenario for interrupts is they all go off at (nearly) the same time. The buffer needs to cover burst or peek behavior and should *never* overflow. However, overflow can be seen during hardware failures which produce a rapid burst of interrupts into the system or unexpectedly high traffic on a particular device. How to handle overflow is dependent upon the semantics of the operation. Sometimes, *e.g.* handling communications packets, data can be dropped and the communications protocol handles the necessary retries. Other system semantics are such that the inability to make background notifications implies a *panic* condition, from which there may be little recourse for recovery. It is necessary to characterize the dynamics of the system when it is functionally complete to tune the buffer memory allocation. This can be said about many aspects of any microcontroller system.

By default, 512 bytes is allocated to the buffer which is 25 slots[4] for the default notification size.

```
<<bg req queue: constants>>=
#ifndef SYS_BG_QUEUE_SIZE
#   define SYS_BG_QUEUE_SIZE 512
#endif /* SYS_BG_QUEUE_SIZE */
```

```
<<bg req queue: static data definitions>>=
BIP_DEFINE_BUFFER(sys_bg_notifications, SYS_BG_QUEUE_SIZE) ;
```

The process of queuing a background notification has three steps:

1. “Probe” the buffer requesting a slot of some given size.
2. Write the notification into the allocated buffer slot.
3. “Push” the slot containing the notification into the buffer. Only after the push, will the notification be visible to any reader of the buffer.

The bip-buffer implementation used here requires that a *push* follow a *probe* in strict order. Probing twice, without an intervening push, always fails except for the case where the previous probe may have failed for other reasons (*e.g.* the buffer does not contain sufficient space for the requested slot size).

```
<<bg req queue: external function declarations>>=
extern void *
probe_bg_queue(
    size_t req_size) ;
```

req_size
: The number of bytes required to hold the background request. If successfully allocated, the buffer slot consists of `req_size` bytes. The memory is guaranteed to be aligned to at least an `size_t` object.

The `probe_bg_queue` function requests the allocation of `req_size` bytes of contiguous memory space in the background notification queue. The return value is a pointer to the allocated notification slot. A return value of `NULL` indicates the buffer is full or the probe request is inappropriate (*e.g.* a previously probed slot has not been pushed).

[4] There is one word of storage used internally by the buffer functions for each requested slot. So 512 bytes is not 32 slots, but 25 plus a bit.

The implementation is trivial and serves only to encapsulate the bip-buffer variable.

Implementation

```
<<bg req queue: external function definitions>>=
void *
probe_bg_queue(
    size_t req_size)
{
    return bip_probe(&sys_bg_notifications, req_size) ;
}
```

After obtaining a slot in the notification queue, the IRQ handler code copies the request into the buffer. Note that the callback function and its parameters which constitute the notification are **passed by value** in a presumed shallow copy. If any of the values passed in the queue are pointer types, then the lifetime of the memory pointed to by the queue value must be managed between the peripheral device code and its proxy function in the background which receives the callback. It is advised that memory allocation and management remain with the background processing and any memory required for buffering by a device be passed in by reference using a device realm background request. This technique is demonstrated in later parts of the book. This is not a hard, fast rule and some exceptions are justifiable (*e.g.* the UART device).

After the notification is copied, it is necessary to "push" it to make it visible in the queue to a reader. All successful invocations of `probe_bg_queue()` must be matched with a call to `push_bg_queue()`.

```
<<bg req queue: external function declarations>>=
extern bool
push_bg_queue(void) ;
```

The `push_bg_queue` function makes the notification allocated by the last invocation of `probe_bg_queue()` available to readers of the notification buffer.

Again the implementation is trivial and serves to encapsulate bip-buffer variable.

Implementation

```
<<bg req queue: external function definitions>>=
bool
push_bg_queue(void)
{
    return bip_push(&sys_bg_notifications) ;          // ❶
}
```

❶ Note it is *not* necessary to supply any sizing information to the `bip_push` operation. Internally, the bip-buffer implementation records the size of allocated slots and once probed, space cannot be returned during the push.

Important

The use of separate functions to probe the queue for request space and then push the populated request buffer is dependent upon setting the same priority for all IRQ's. By doing so, there is no possible preemption that might take place between the probe and the push. A nested interrupt configuration needs a slightly different design where the probe and push happen in a section of code with the execution priority raised to prevent preemption. The two function design allows the caller to build the notification directly into the allocated buffer slot before it is pushed. The following function shows an implementation of the probe and push in a single function. This function could form the basis for one which would work properly in a nested interrupt scenario. In all cases, careful consideration of the preemption possibility is required. The bip-buffer implementation does not allow improperly ordered probe / push sequences. In this case, the flat IRQ priority scheme yields a small amount of flexibility in structuring the operation.

For convenience, those IRQ handlers which find it more convenient to build the notification in a local variable (*e.g.* the notification may be constant), can use the `send_bg_notification` function to perform the probe/write/push sequence in one step.

```
<<bg req queue: external function declarations>>=
extern bool
send_bg_notification(
    SVC_DevNotification const *const notify) ;
```

notify
A pointer to a device notification block which is to be placed in the background notification queue.

The `send_bg_notification` function inserts a background notification into the background notification queue as a single operation. For simple cases where the notification information is already present in a variable, this function performs the probe, write, push sequence as one function. Return `true` if the write to the background notification queue is successful and `false` otherwise.

Implementation

```
<<bg req queue: external function definitions>>=
bool
send_bg_notification(
    SVC_DevNotification const *const notify)
{
    bool written = false ;
    void *notify_slot = probe_bg_queue(notify->block_size) ;
    if (notify_slot != NULL) {
        memcpy(notify_slot, notify, notify->block_size) ;
        written = push_bg_queue() ;
    }

    return written ;
}
```

The following function is used when notifications are retrieved from the queue and copied to the background.

```
<<bg req queue: external function declarations>>=
extern int
copy_out_bg_notification(
    size_t buffer_size,
    void *buffer,
    size_t *const next_size_ref) ;
```

buffer_size
The number of bytes for the memory object pointed to by `buffer`.

buffer
A pointer to a memory object where the background request is placed. `buffer` is assumed to point into unprivileged memory.

next_len
A pointer to a memory object where the length of the next background notification is placed.

The `copy_out_bg_notification` function removes the first background notification from the read side of the background notification queue. The return value is 0 if the queue is empty. A positive return value indicates the number of bytes copied into `buffer`. A negative error return value indicates that the number of bytes in the request exceeded `buffer_size`. If a negative value is returned, the background notification queue remains unmodified. By special dispensation, if invoked with the value of `buffer_size` as zero or `buffer` as `NULL` no data is transferred but the value of the variable pointed to by `next_size_ref` is set to the size of the notification at the head of the queue.

Implementation

```
<<bg req queue: external function definitions>>=
int
copy_out_bg_notification(
    size_t buffer_size,
    void *buffer,
    size_t *const next_size_ref)
{
    assert(buffer != NULL || buffer_size == 0) ;
    assert(next_size_ref != NULL) ;

    int result = 0 ;
    size_t req_size ;
    void *request = bip_peek(&sys_bg_notifications, &req_size) ;
    if (request != NULL && buffer != NULL && req_size <= buffer_size) {
        memcpy_to_unpriv(buffer, request, req_size) ;
        result = (int)req_size ;
        (void)bip_pop(&sys_bg_notifications, next_size_ref) ;
    } else {
        *next_size_ref = req_size ;
    }

    return result ;
}
```

Watchdog Timer

This section shows the device code for a watchdog timer. As a first demonstration of device control, a simple peripheral is illustrative. A watchdog timer is nonetheless an important peripheral that is needed as part of the basic protections for any system. In later chapters, many other examples of peripherals which generate interrupts and issue background notifications are shown.

A watchdog timer is a device which helps to insure that some fundamental function of a system is operational by repeatedly requiring the system to *service* (or *acknowledge* or *feed* or *pet*) the watchdog. Typically, a watchdog timer is used to insure that overall execution flow in the system is functioning. Should some code sequence result in a long compute bound, possibly infinite loop, then other operations in the system may be starved for access to the CPU and essential functions of the system might not execute. Sometimes a watchdog is used to insure that a particular system function, in contrast to the system as a whole, works correctly. Such system functions are usually periodic and a watchdog may be used to insure that the function happens at the correct period. For example, cardiac pacemakers have a watchdog timer that specifically monitors the issuing of pace pulses to the heart. Failure to issue a pace pulse regularly and within a certain time frame would be a fundamental system failure and be hazardous to the patient. Potentially, the pacemaker could fail to issue pace pulses yet its overall control flow might still be functional. Some SOC's supply multiple watchdog timers to allow monitoring overall system execution as well as specialized system operations.

Watchdog timers peripherals work by counting the cycles of a clock. If the timer count ever reaches 0 (if counting down) or a specified value (if counting up), then the watchdog issues a system reset signal. Software is expected to service the watchdog to restart its counting. The specifics of watchdog capabilities varies between SOC's, but most watchdog timer peripherals usually include a means for the watchdog to notify software that the reset period expires soon. This takes the form of an interrupt to prompt the system software to service the watchdog. This gives software a maximum deadline in which it must accomplish the watchdog service. Some watchdog timers also have the facility to cause a reset if the timer is serviced too soon, *i.e.* it demands servicing happen in a time window before the reset time. There are still other variations on the theme.

It is particularly important that code which services the watchdog exercise the critically important paths that are being protected. This implies that you *must not* restart the watchdog in the IRQ handler associated with a watchdog notification interrupt. This would negate any protection offered by a watchdog timer. It is easy to conceive of an infinite loop running somewhere that is preempted by the watchdog interrupt but is otherwise preventing essential system operations.

The figure below, taken from the Apollo 3 datasheet, shows the logic of the particular watchdog timer peripheral for the code shown here.

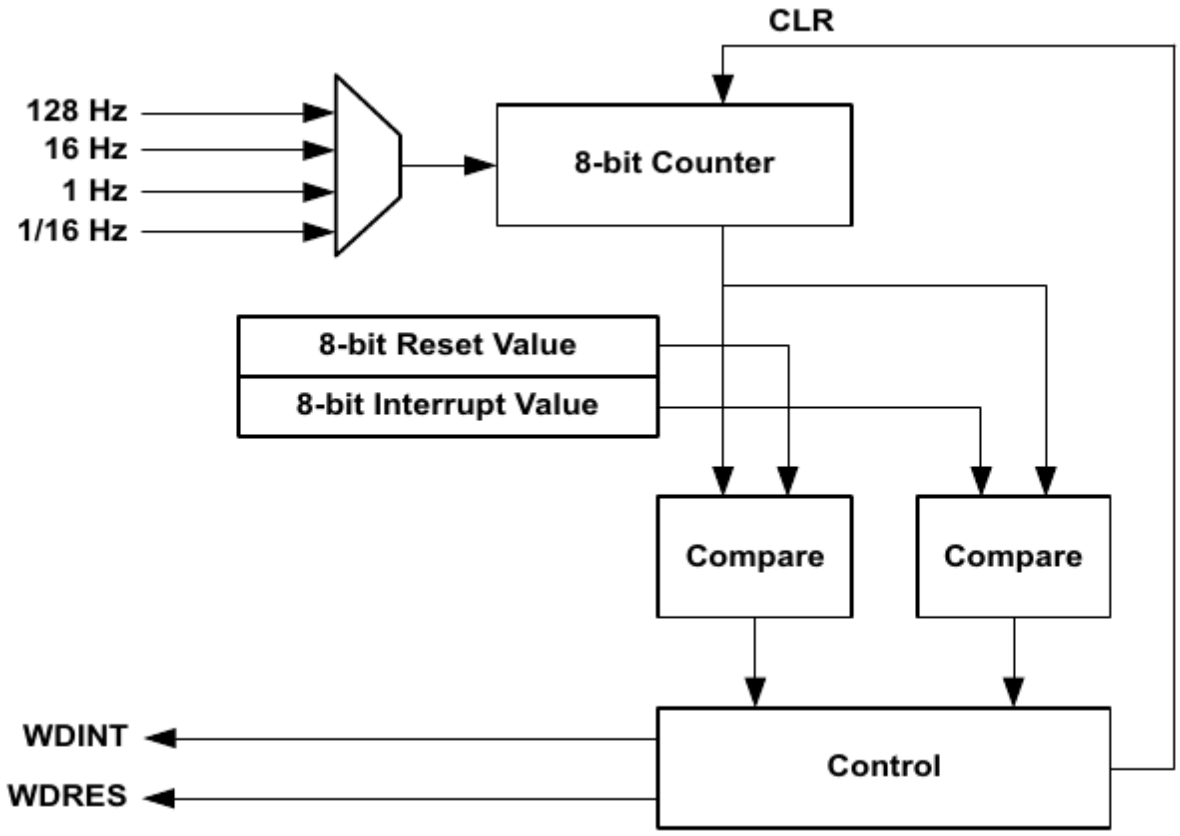

Figure 4.5: Schematic of Apollo 3 Watchdog Timer Peripheral

This peripheral operates by using a selectable clock to increment a counter. The incrementing counter can trigger a comparison match against the values of two compare registers. A match between the values in the `Counter` and `Interrupt Value` registers causes an interrupt. A match between the `Counter` and `Reset Value` registers causes a system reset.

SOC designers parameterize the operations of peripheral devices to make them adaptable to a wider range of applications. The goal here is to tailor the usage of the devices to the particular requirements of the system being built. There is no intent to support every possible configuration. For example, the Apollo 3 watchdog timer supports *locking* the configuration registers so that once set, they may not be changed until the system is reset. The paranoia level does not extend to the need to prevent highly improbable register writes, especially as that possibility has been excluded from the background processing as a result of separating privileged and unprivileged execution.

One quirk of the Apollo 3 watchdog does require some special arrangements and accommodation. Many SOC watchdog timers have a setting which disables the watchdog timer clock when a debugger is active. This makes breakpoint style debugging much easier. The Apollo 3 watchdog does *not* have this ability. It is a pain point that requires a work around.

When designing the interface to a peripheral device, a drawing is beneficial, even for a device as simple as a watchdog timer. The following figure shows the components of the watchdog timer interface.

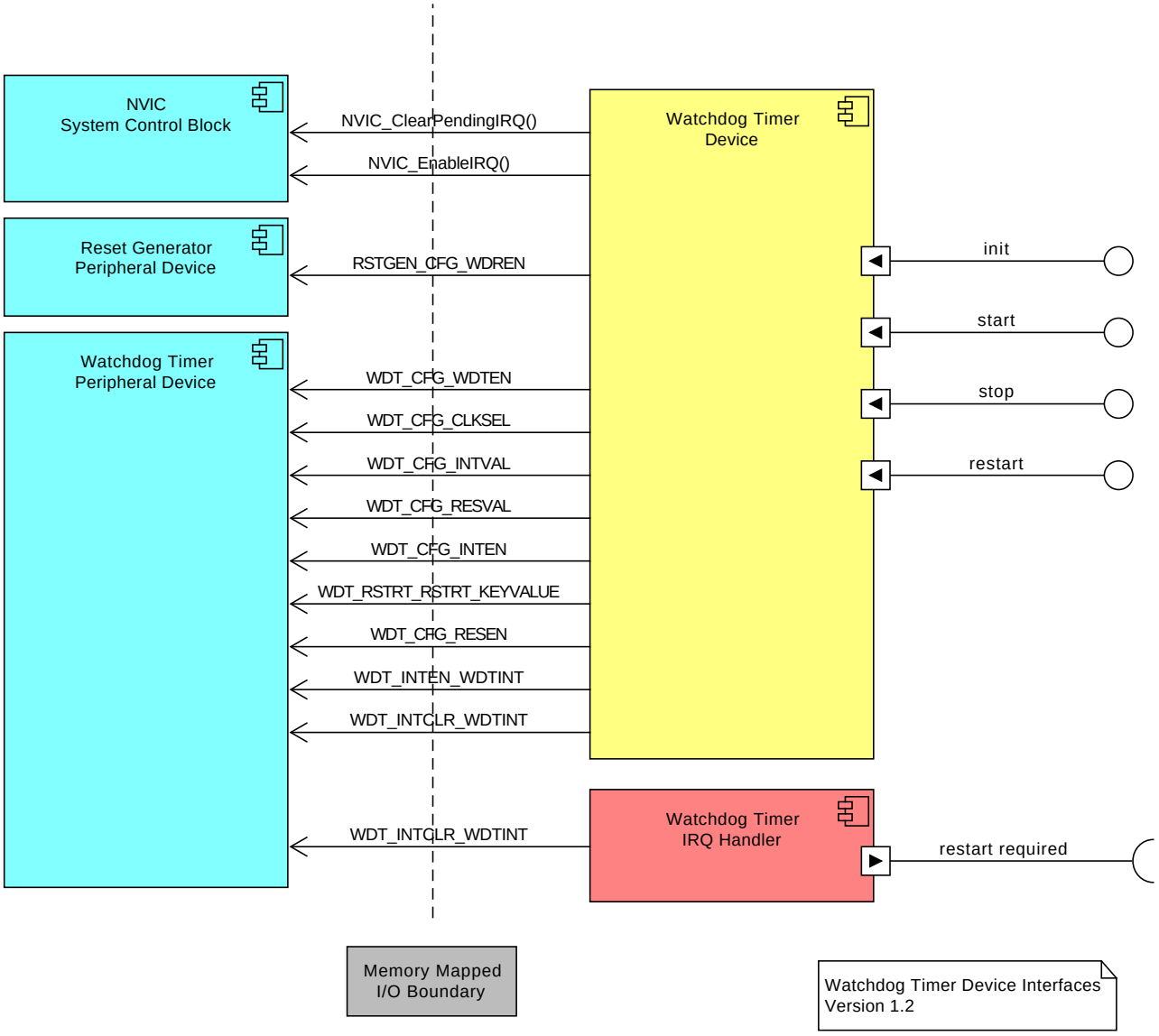

Figure 4.6: Watchdog Timer Interface Components

This figure repurposes some UML graphical symbols to show the interactions between the watchdog timer background requests and the underlying peripheral devices. The yellow component represents the collection of device realm foreground proxy functions provided by the watchdog timer device. The available functions are:

init
: Configure the device and make it ready to run.

start
: Start the watchdog timer running.

stop
: Stop the watchdog timer.

restart
: Reset the watchdog timer to zero. This operation acknowledges the watchdog preventing it from causing a system reset.

The red component represents the watchdog timer IRQ handler. It issues a single background notification indicating that the watchdog expires soon and needs to be restarted.

The cyan components represent Apollo 3 peripheral devices. The arrows represent access (read or write) to the peripheral registers or register fields as given by the label.

NVIC
The NVIC is used to globally clear and enable the interrupt.

Reset Generator
For the watchdog timer to cause a system reset, it must be enabled to do so at the reset generator peripheral.

Watchdog Timer
The watchdog timer peripheral has a set of registers to control its functions.

Watchdog Timer Requests

There are four functions provided to background processing to control the watchdog timer. This implies that eight functions must be provided, four used by background processing to make watchdog requests and four that serve as proxies to the background requests. The sections below cover each watchdog timer operation.

Init

It is necessary to initialize the watchdog timer before use.

```
<<dev svc wdog req: external declarations>>=
extern int
dev_wdog_init(
    SVC_DevInstance wdog_instance,
    WDOG_TimeTicks wdog_reset_time,
    WDOG_TimeTicks wdog_notify_time,
    SVC_DevNotifyProxy notify_proxy,
    SVC_DevNotifyClosure notify_closure) ;
```

wdog_instance
The instance of the watchdog timer on which the initialization operation is performed.

wdog_reset_time
The amount of time that must elapse *without* the watchdog timer being restarted before the system is reset. The `reset_time` is an unsigned UQ4.4 fixed binary point value in units of seconds and must be greater than zero.

wdog_notify_time
The amount of time which elapses before the watchdog timer issues a notification that a watchdog restart is necessary to prevent a system reset. The `notify_time` is an unsigned UQ4.4 fixed binary point value in units of seconds which must be greater than zero and less than `reset_time`.

notify_proxy
A pointer to a function that is to be invoked during background processing when the watchdog timer notification is dispatched.

notify_closure
A value passed to `notify_proxy` when it is dispatched. The value is large enough to hold a pointer. If a pointer value is used, its type must be recovered in the proxy function and the lifetime of memory pointed to is the responsibility of the caller (*i.e.* if `notify_closure` is a pointer, background processing must insure the memory is still valid by the time the notification is dispatched, *e.g.* using a pointer to an automatic variable is undefined). If no additional closure information is needed by `notify_proxy`, it is recommended that `notify_closure` have a value of zero.

The `dev_wdog_init` function initializes the watchdog timer instance given by the `wdog_inst` argument. This function must be called before any other watchdog timer function.

The time arguments have a range from approx 0.0625 to 15.9 seconds. After starting the watchdog timer using `dev_wdog_start()`, the difference between the `wdog_reset_time` and `wdog_notify_time` is the amount of time software has to invoke the `dev_wdog_restart()` function before the watchdog timer causes a system reset.

A return value of 0 indicates success. A negative return value indicates a failure.

Note that the function provides an instance argument despite there being only a single instance of watchdog timer in the Apollo 3. It is included in the interface for consistency and future compatibility.

The units for the notify and reset times are raw device units. To manage the units conversions, a typedef and utility functions are supplied.

```
<<dev realm wdog param: data type declarations>>=
typedef uint8_t WDOG_TimeTicks ;
```

The conversion from milliseconds to watchdog time units is accomplished by the following function.

```
<<dev svc wdog req: external declarations>>=
extern WDOG_TimeTicks
dev_util_wdog_ms_to_ticks(
    uint32_t ms) ;
```

msec
A number of milliseconds to convert into watchdog timer ticks.

The `dev_wdog_msec_to_ticks` function converts the `msec` value into watchdog timer ticks in the form of an unsigned UQ4.4 fixed binary point number. The maximum amount of time which can be specified is 15.9375 seconds. The minimum time amount is 0.0625. Any value of `ms` which causes an overflow of the allowed range is silently truncated to the maximum time value. Any value of `ms` which converts to zero ticks returns 1.

```
<<dev svc wdog req: external definitions>>=
WDOG_TimeTicks
dev_util_wdog_ms_to_ticks(
    uint32_t ms)
{
    uint32_t ticks = ((ms << 4) + UINT32_C(500)) / UINT32_C(1000) ; // ❶
    ticks = min(ticks, UINT32_C(0xff)) ;
    ticks = max(ticks, UINT32_C(1)) ;

    return (WDOG_TimeTicks)ticks ;
}
```

❶ Using 32 bits to avoid overflow.

The following data structure is used to marshal the arguments to `dev_wdog_init` for passing to the foreground proxy function.

```
<<dev realm wdog param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevWdogInitInput,
    WDOG_TimeTicks wdog_reset_time ;
    WDOG_TimeTicks wdog_notify_time ;
    SVC_DevNotifyProxy notify_proxy ;
    SVC_DevNotifyClosure notify_closure ;
) ;
```

Only input parameters are required for initializing the watchdog.

Implementation

```
<<dev svc wdog req: external definitions>>=
int
dev_wdog_init(
    SVC_DevInstance wdog_instance,
    WDOG_TimeTicks wdog_reset_time,
    WDOG_TimeTicks wdog_notify_time,
    SVC_DevNotifyProxy notify_proxy,
    SVC_DevNotifyClosure notify_closure)
{
    SVC_DevWdogInitInput wd_input = {
        .block_size = sizeof(SVC_DevWdogInitInput),
        .wdog_reset_time = wdog_reset_time,
        .wdog_notify_time = wdog_notify_time,
        .notify_proxy = notify_proxy,
        .notify_closure = notify_closure,
    } ;

    SVC_DevRequest wdog_req = dev_req_encode(DEV_WDOG_CLASS, wdog_init, wdog_instance) ;
```

```
    return dev_realm_svc_call(wdog_req, &wd_input, NULL, NULL) ;
}
```

During initialization, the information used to issue a notification to the background when the watchdog timer interrupt occurs must be saved. The notification for a simple device like this can use the generic notification structure. Since there is only one instance of the watchdog timer, the information can be saved in a single variable.

```
<<dev realm wdog proxy: static data definitions>>=
static SVC_DevNotification wdog_notification ;
```

The following function is the foreground proxy function for `dev_wdog_init`. This function receives control from the `SVC` exception handler indirectly through the watchdog timer operation handler.

```
<<dev realm wdog proxy: static function definitions>>=
static int
dev_realm_wdog_init(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    (void)req ;

    SVC_DevWdogInitInput wd_input ;
    int result = copy_in_svc_param(input, sizeof(wd_input), &wd_input) ;
    if (result < 0) {
        return result ;
    }

    if (wd_input.notify_proxy == NULL ||
            wd_input.wdog_reset_time == 0 ||
            wd_input.wdog_notify_time == 0 ||
            wd_input.wdog_reset_time <= wd_input.wdog_notify_time) {
        return -ERR_INVALID_PARAM ;
    }

    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    NVIC_DisableIRQ(WDT_IRQn) ;                                          // ❶
    BTWD_CLEAR_REG_FIELD(&WDT->CFG, WDT_CFG_WDTEN) ;

    wdog_notification.block_size = sizeof(wdog_notification) ;
    wdog_notification.device_class = dev_req_extract_class(req) ;
    wdog_notification.device_instance = dev_req_extract_instance(req) ;
    wdog_notification.notify_proxy = wd_input.notify_proxy ;
    wdog_notification.notify_closure = wd_input.notify_closure ;

    uint32_t cfg = 0 ;
    cfg = BTWD_FIELD_INSERT(cfg, WDT_CFG_CLKSEL_16HZ, WDT_CFG_CLKSEL) ;
    cfg = BTWD_FIELD_INSERT(cfg, wd_input.wdog_notify_time, WDT_CFG_INTVAL) ;
    cfg = BTWD_FIELD_INSERT(cfg, wd_input.wdog_reset_time, WDT_CFG_RESVAL) ;
    cfg = BTWD_FIELD_SET(cfg, WDT_CFG_INTEN) ;
    WDT->CFG = cfg ;

    BTWD_SET_REG_FIELD(&WDT->INTCLR, WDT_INTCLR_WDTINT) ;
    BTWD_SET_REG_FIELD(&WDT->INTEN, WDT_INTEN_WDTINT) ;
    NVIC_ClearPendingIRQ(WDT_IRQn) ;
    NVIC_EnableIRQ(WDT_IRQn) ;

    return 0 ;
}
```

❶ In case the watchdog is re-initialized, prevent any interrupt and disable the timer.

Start

```
<<dev svc wdog req: external declarations>>=
extern int
dev_wdog_start(
    SVC_DevInstance wdog_instance,
    bool enable_reset) ;
```

wdog_instance
The instance of the watchdog timer which is to be started.

enable_reset
A boolean value which determines whether system is reset when the watchdog timer expires. A value of `true` implies that an unacknowledged watchdog causes a system reset. A value of `false` implies that the watchdog timer never causes a system reset. The use case for this parameter is to allow breakpoint style debugging since the Apollo 3 SOC does *not* halt the watchdog timer clock when stopped for debugging purposes.

The `dev_wdog_start` function starts the watchdog timer running. This function must be called only after invoking `dev_wdog_init`. After invoking `dev_wdog_start`, background notifications will be issued periodically according to the value of `wdog_notify_time` given to the `dev_wdog_init` function.

A return value of 0 indicates success. A negative return value indicates a failure.

The single argument, `enable_reset` is given as an input to the proxy function.

Implementation

```
<<dev realm wdog param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevWdogStartInput,
    bool enable_reset ;
) ;
```

```
<<dev svc wdog req: external definitions>>=
int
dev_wdog_start(
    SVC_DevInstance wdog_instance,
    bool enable_reset)
{
    SVC_DevWdogStartInput request = {
        .block_size = sizeof(SVC_DevWdogStartInput),
        .enable_reset = enable_reset,
    } ;

    SVC_DevRequest wdog_req = dev_req_encode(DEV_WDOG_CLASS, wdog_start, wdog_instance) ;
    return dev_realm_svc_call(wdog_req, &request, NULL, NULL) ;
}
```

The proxy function implementation follows the pattern established by `dev_realm_wdog_init`.

```
<<dev realm wdog proxy: static function definitions>>=
static int
dev_realm_wdog_start(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
```

```
{
    (void)req ;

    SVC_DevWdogStartInput wd_input ;
    int result = copy_in_svc_param(input, sizeof(wd_input), &wd_input) ;
    if (result < 0) {
        return result ;
    }

    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    (void)stwd_begin_interrupt_section(WDT_IRQn) ;          // ❶
//BEGIN INTERRUPT SECTION

    BTWD_CLEAR_REG_FIELD(&WDT->CFG, WDT_CFG_WDTEN) ;

    if (wd_input.enable_reset) {
        BTWD_SET_REG_FIELD(&WDT->CFG, WDT_CFG_RESEN) ;
        BTWD_SET_REG_FIELD(&RSTGEN->CFG, RSTGEN_CFG_WDREN) ;
    } else {
        BTWD_CLEAR_REG_FIELD(&WDT->CFG, WDT_CFG_RESEN) ;
        BTWD_CLEAR_REG_FIELD(&RSTGEN->CFG, RSTGEN_CFG_WDREN) ;
    }

    WDT->RSTRT = WDT_RSTRT_RSTRT_KEYVALUE ;
    WDT->INTCLR = WDT_INTCLR_WDTINT_Msk ;
    BTWD_SET_REG_FIELD(&WDT->CFG, WDT_CFG_WDTEN) ;

    stwd_end_interrupt_section(WDT_IRQn, true) ;
//END INTERRUPT SECTION

    return 0 ;
}
```

❶ The foreground proxy functions run at `SVC` exception priority. To allow starting an already running watchdog, it is safer to prevent the watchdog timer interrupt from happening since it would preempt the foreground proxy.

Stop

```
<<dev svc wdog req: external declarations>>=
extern int
dev_wdog_stop(
    SVC_DevInstance wdog_instance) ;
```

wdog_instance
The instance of the watchdog timer which is to be stopped.

The `dev_wdog_stop` function stops the watchdog timer given by `wdog_instance`. After invoking `dev_wdog_stop` no further notifications from the device are issued until it is started again.

A return value of 0 indicates success. A negative return value indicates a failure.

There are no parameters for the stop proxy function.

Implementation

```
<<dev svc wdog req: external definitions>>=
int
dev_wdog_stop(
    SVC_DevInstance wdog_instance)
{
    SVC_DevRequest wdog_req = dev_req_encode(DEV_WDOG_CLASS, wdog_stop, wdog_instance) ;
    return dev_realm_svc_call(wdog_req, NULL, NULL, NULL) ;
}
```

```
<<dev realm wdog proxy: static function definitions>>=
static int
dev_realm_wdog_stop(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    (void)req ;
    assert(input == NULL) ; (void)input ;
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    (void)stwd_begin_interrupt_section(WDT_IRQn) ;
//BEGIN INTERRUPT SECTION

    BTWD_CLEAR_REG_FIELD(&WDT->CFG, WDT_CFG_WDTEN) ;
    BTWD_CLEAR_REG_FIELD(&WDT->INTCLR, WDT_INTCLR_WDTINT) ;
    BTWD_CLEAR_REG_FIELD(&RSTGEN->CFG, RSTGEN_CFG_WDREN) ;

    stwd_end_interrupt_section(WDT_IRQn, true) ;
//END INTERRUPT SECTION

    return 0 ;
}
```

Restart

```
<<dev svc wdog req: external declarations>>=
extern int
dev_wdog_restart(
    SVC_DevInstance wdog_instance) ;
```

wdog_instance
The instance of the watchdog timer which is to be restarted.

The `dev_wdog_restart` function restarts the watchdog timer given by `wdog_instance` insuring that the watchdog does not cause a system reset.

There are no parameters for the restart proxy function.

Implementation

```
<<dev svc wdog req: external definitions>>=
int
dev_wdog_restart(
    SVC_DevInstance wdog_instance)
{
```

```
    SVC_DevRequest wdog_req = dev_req_encode(DEV_WDOG_CLASS, wdog_restart, wdog_instance) ;
    return dev_realm_svc_call(wdog_req, NULL, NULL, NULL) ;
}
```

The implementation of the proxy function is just as simple. A single register write is sufficient.

```
<<dev realm wdog proxy: static function definitions>>=
static int
dev_realm_wdog_restart(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    (void)req ;
    assert(input == NULL) ; (void)input ;
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    WDT->RSTRT = WDT_RSTRT_RSTRT_KEYVALUE ;

    return 0 ;
}
```

Dispatching Watchdog Requests

The device realm has an additional level of dispatch and a function is required to find the proxy function associated with the device operation.

```
<<dev realm wdog param: constants>>=
#define WDOG_INSTANCES   1
```

```
<<dev realm wdog proxy: external declarations>>=
extern int
dev_realm_wdog(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

An enumeration is used to encode the watchdog operations. *N.B.* it is important to encode the operations as zero based sequential integers. The values are used as array indices.

```
<<dev realm wdog param: data type declarations>>=
enum SVC_wdogOperation {
    wdog_init = 0,
    wdog_start,
    wdog_stop,
    wdog_restart,

    wdog_operation_count        // last
} ;
```

Like the service realm dispatch function, operation dispatching for a device is also based on a jump table.

Implementation

```
<<dev realm wdog proxy: external function definitions>>=
int
dev_realm_wdog(
```

```
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    int status = dev_req_validate(req, wdog_operation_count, WDOG_INSTANCES) ;  // ❶
    rtcheck_zero_return(status, status) ;

    static const SVC_DevRequestProxy wdog_proxies[wdog_operation_count] = {
        [wdog_init] = dev_realm_wdog_init,
        [wdog_start] = dev_realm_wdog_start,
        [wdog_stop] = dev_realm_wdog_stop,
        [wdog_restart] = dev_realm_wdog_restart,
    } ;

    SVC_DevOperation wdog_operation = dev_req_extract_operation(req) ;
    return wdog_proxies[wdog_operation](req, input, output, error) ;
}
```

❶ The next step in validating the `req` parameter is done at the device operation dispatch level. Here enough information is known to insure the operation encoding and instance encoding passed across the SVC interface is correct.

The watchdog timer device must be assigned a class identifier and a pointer to the operation dispatch function must be placed in the jump table for device realm dispatch.

```
<<dev realm param: constants>>=
#define DEV_WDOG_CLASS        0
```

```
<<svc entry: device request classes>>=
[DEV_WDOG_CLASS] = dev_realm_wdog,
```

Watchdog Timer IRQ Handler

The last part of the watchdog timer device code is the IRQ handler. This is the first example that shows how an IRQ handler posts background notifications. This notification informs background processing that it should service the watchdog device soon or else the system may reset (depending upon how the watchdog was started). This notification exercises a critical part of the system control flow which is discussed in the next section.

Important
This is only part of the overall execution path that needs to be exercised. How the notification is handled by background processing is essential to closing the monitoring loop that insures the system behaves within expected bounds.

During initialization (as shown previously in `dev_wdog_realm_init`), necessary information was stored away for later use in formulating the notification from the device. The implementation of the IRQ Handler uses the previously saved notification information and inserts it into the background notification queue. Note it is not necessary to protect access to the background request queue since an IRQ handler is running and all IRQ's have the same priority. No other IRQ can preempt the execution and no higher priority exception accesses the background notification queue.

```
<<dev realm wdog proxy: external function definitions>>=
void
WDT_IRQHandler(void)
{
    BTWD_SET_REG_FIELD(&WDT->INTCLR, WDT_INTCLR_WDTINT) ;
```

```
    assert (wdog_notification.notify_proxy != NULL) ;
    if (wdog_notification.notify_proxy != NULL) {                          // ❶
        bool sent = send_bg_notification(&wdog_notification) ;
        if (!sent) {
            panic("failed to post watchdog background notification") ;     // ❷
        }
    } else {                                                               // ❸
        NVIC_DisableIRQ(WDT_IRQn) ;
        BTWD_CLEAR_REG_FIELD(&WDT->CFG, WDT_CFG_WDTEN) ;
    }
}
```

❶ Another check to insure that the `NULL` pointer value is not pushed into the background processing.

❷ The background notification queue is one of the critical system functions checked by the watchdog notification scheme. Failing to post the background notification is definitely a "panic" situation and failing here, before the watchdog device causes a reset, gives additional information for possible diagnosis.

❸ Note if somehow the IRQ Handler has a `NULL` notification proxy function, the watchdog timer is disabled altogether. Disabling the watchdog might lead to a reset, but something is already very wrong if the interrupt is enabled and there is no background notification proxy function.

As this is the first IRQ Handler shown, note that there is no additional processing required upon entry into or exit from the IRQ Handler. The vector table contains a direct pointer to the function just shown. There is no need to inform the execution architecture that an IRQ has occurred. The only operation an IRQ Handler can perform which has an effect on the system is to post a background notification. An IRQ Handler may *not* need to post a notification or may do so conditionally.

Foreground to Background Control Flow

This chapter has described techniques to build a system control mechanism to mediate between foreground processing and background processing. To recap, execution is initiated by interrupt requests which come from conditions detected in the environment as sensed by peripheral devices. The peripheral device control code generates notifications to background processing indicating the changes in the environment. These notifications are placed in the **Background Notification Queue**. The sole control transfer mechanism between foreground and background processing is for the background run time to repeatedly request the next notification and then dispatch it. When the notification queue is empty and when background processing has determined there is no additional work to perform as a result of having dispatched all the notifications, it issues a request to the foreground to *wait* until an IRQ handler posts a new notification indicating actionable conditions that have arisen in the environment.

There are a couple of ideas that need reiteration.

- The mechanism provided here is strictly to allow *injecting* conditions detected in the environment into background processing so they may be acted upon.
- This is *not* the only control mechanism needed. How background processing is to handle the notifications it receives has *not* been presented. That is another interesting topic and Part II of this book is devoted to explaining the background processing execution scheme. For now, working up from the bottom, the focus is only on the mechanism used to have foreground processing be responsive to the environment and to inform background processing of what has happened. As you might guess, since a queue was involved in the transport of notifications to the background, a queue is used in the background to create the system reaction to those notifications.

Retrieving Notifications into the Background

As discussed previously, when background processing makes a request for system or device realm service, those requests are directed to a proxy function which executes the request for the background. Both sides must agree on the exact structure of data sent between the functions and the exact semantics of the operation performed. The `SVC` exception is used to "make the

leap" from background to foreground processing. The flow of control from background to foreground is synchronous and runs to completion. Although the execution flow from background through to foreground proxy may be preempted by an IRQ, from the point of view of the background it appears as a conventional synchronous function invocation, *i.e.* nothing else goes on in the background until the foreground proxy function returns.

Similarly, when an IRQ hander detects an actionable condition, it issues a notification to the background. The notification requests background processing to execute a proxy function which handles the notification. The notification proxy function executed is one that was configured previously by background processing. In the following sections, the specific operations and interfaces needed by background processing to obtain foreground notifications are presented. Ultimately, it background processing that dispatches the notification proxy function to execute as part of the background reaction to the notification.

The following function, executed in the background, retrieves a notification.

```
<<sys svc req: external declarations>>=
extern int
sys_get_bg_notification(
    size_t buffer_size,
    void *buffer,
    size_t *next_size_ref) ;
```

buffer_size
: The number of bytes of memory pointed to by `buffer`.

buffer
: A pointer to a memory object of at least `buffer_size` bytes where the next background notification is placed.

next_size_ref
: A pointer to a memory object where the size of the next background notification request is returned. If `next_size_ref` is NULL, then the next notification length is not returned. A returned next length of 0 implies that there are no other notification in the queue.

The `sys_get_bg_notification` obtains the next background notification queued by an IRQ handler. The request is placed in the memory pointed to by `buffer` whose size is given by `buffer_size`. The return value of the function is an integer. The interpretation of the returned integer is:

less than 0
: an error, as indicated by the negative return value, occurred.

equal to 0
: the background notification queue is empty, no copy took place.

greater than 0
: the notification was copied into `buffer` and the returned value is the number of bytes placed in `buffer`.

For those cases where `buffer_size` is too small to hold the next background notification in the queue, an error is returned and the background notification queue is not modified. It is allowed to invoke `sys_get_bg_notification` with a zero `buffer_size` value. For that case no transfer happens, but it can be used to obtain the size of the notification at the head of the queue.

Following the pattern of the system realm interface discussed previously, a data structure is used to pass the parameters of `sys_get_bg_notification` to its foreground proxy. In this case, the foreground proxy needs input parameters for the `buffer` pointer value and its `buffer_size` length value.

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysGetBGRequestInput,
    size_t buffer_size ;
    void *req_buffer ;
) ;
```

An output parameter is needed to return the number of bytes actually copied into `buffer` and the size of the next notification in the queue. *N.B.* input parameters are considered as `const` and an input value may *not* be overwritten with an output value.

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysGetBGRequestOutput,
    size_t copied ;
    size_t next_notify_len ;
) ;
```

The error parameter is not used.

The implementation of `sys_get_bg_notification` fills in the request input and output blocks and invokes the function for a system realm request. Do not be confused by using a system realm request to retrieve a device realm notification. The notification queue is a *system realm* resource despite holding *device realm* information.

Implementation

```
<<sys svc req: external definitions>>=
int
sys_get_bg_notification(
    size_t buffer_size,
    void *buffer,
    size_t *next_size_ref)
{
    SVC_SysGetBGRequestInput const req_input = {
        .block_size = sizeof(SVC_SysGetBGRequestInput),
        .buffer_size = buffer_size,
        .req_buffer = buffer,
    } ;
    SVC_SysGetBGRequestOutput req_output = {
        .block_size = sizeof(SVC_SysGetBGRequestOutput),
        .copied = 0,
        .next_notify_len = 0,
    } ;

    int status = sys_realm_svc_call(SYS_GET_BG_NOTIFICATION,
            &req_input, &req_output, NULL) ;
    rtcheck_zero_return(status, status) ;

    if (next_size_ref != NULL) {
        *next_size_ref = req_output.next_notify_len ;
    }

    return (int)req_output.copied ;
}
```

As before, the request type is given an unique identifying number and inserted it into the jump table used for dispatching requests in the system realm.

```
<<sys realm param: constants>>=
#define SYS_GET_BG_NOTIFICATION     1
```

```
<<svc entry: system request functions>>=
[SYS_GET_BG_NOTIFICATION] = sys_realm_get_bg_notification,
```

The foreground proxy function which implements the operation to obtain the next background service request has the following prototype:

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_get_bg_notification(
    SVC_SysRequest req,
```

```
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

The main actions of the foreground proxy function are:

1. Validate the parameters.
2. “Peek” at the background request buffer to determine it there is a pending request.
3. Copy any request into the supplied buffer and “pop” the request buffer to release the space occupied by the now copied request.

Implementation

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_get_bg_notification(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(req == SYS_GET_BG_NOTIFICATION) ; (void)req ;
    assert(error == NULL) ; (void)error ;

    SVC_SysGetBGRequestInput req_input ;
    int result = copy_in_svc_param(input, sizeof(req_input), &req_input) ;
    rtcheck_zero_return(result, result) ;
    rtcheck_return(req_input.req_buffer != NULL || req_input.buffer_size == 0,
            -ERR_INVALID_PARAM) ;

    SVC_SysGetBGRequestOutput req_output = {
        .block_size = sizeof(req_output),
        .copied = 0,
        .next_notify_len = 0,
    } ;

    stwd_begin_priority_section(IRQ_PRIORITY) ;                    // ❶
//BEGIN PRIORITY SECTION

    result = copy_out_bg_notification(req_input.buffer_size,
            req_input.req_buffer, &req_output.next_notify_len) ;

    stwd_end_priority_section() ;
//END PRIORITY SECTION

    if (result >= 0) {
        req_output.copied = (size_t)result ;
        result = copy_out_svc_result(output, sizeof(req_output), &req_output) ;
    }

    return result ;
}
```

❶ Since execution is at SVC priority and since the background notification queue is shared with most IRQ’s, the copy out of the notification must not be preempted by an IRQ. This means raising the current priority to the configured IRQ priority. Other exceptions, most notably the system faults, are allowed to preempt and run.

Waiting for New Background Notifications

The previous section showed the required processing for the background run time to obtain the background notifications posted by IRQ handlers. There is another fundamental primitive which is required to complete the foreground to background control flow. In this section, how background processing waits for new requests when there is no more background work to be done is shown.

When background processing has determined there is no addition work to be done, it waits until an interrupt occurs and a new background request is posted. Following the established pattern, the background function is given followed by the foreground proxy. To emphasize, simply dispatching background notifications is *not* the totality of the work performed by background processing. In whatever manner background processing determines that all the actions started by notifications are completed, it must wait for new notifications to arrive. A new notification arrives only as a result of an IRQ handler placing a notification in the background notification queue.

The fundamental processor instruction for waiting is `WFI`, Wait For Interrupt. On most ARMv7-M SOC's, this causes the processor to go to sleep. Exactly what happens when the processor goes to sleep is not determined by the core, but by the SOC design which uses the core. In the case of the Apollo 3, the processor is put into a sleep mode and various clocks and other peripherals are powered down. Most SOC's use the `WFI` instruction as an indication to turn off the high speed clock to the processor core as the first step in reducing power consumption. The high speed clocks are some of the more power consuming components of an SOC. Some SOC's have more elaborate levels of power reduction that allow power management to be more than a simple binary choice.

The function used by the background run time to wait for new notifications has the following prototype.

```
<<sys svc req: external declarations>>=
extern int
sys_wait(void) ;
```

The `sys_wait()` function requests the processor to cease execution until a background notification has been inserted into the background notification queue.

The implementation is trivial since there are no arguments to the foreground proxy function which implements the wait.

Implementation

```
<<sys svc req: external definitions>>=
int
sys_wait(void)
{
    return sys_realm_svc_call(SYS_WAIT_REQ, NULL, NULL, NULL) ;
}
```

The function prototype for the foreground proxy function has the usual parameters. In this particular case, only the `req` parameter is of interest and the three parameter pointers are passed `NULL`.

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_wait(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

The implementation of `sys_realm_wait()` has several circumstances that it must accommodate.

- There is a *race condition* between when the background processing determines that the background notification queue is empty and when the foreground proxy code for waiting executes. An interrupt can occur between those times. Typically, when this function is invoked, the background notification queue has been emptied. Therefore, before sleeping it is necessary to check that the queue remains empty of any pending notifications.

- The `WFI` instruction has a set of conditions[5] which must be met before the processor core wakes back up once it has been put to sleep. For this use case, the situation when there is a pending exception whose priority is such that it would preempt any currently active exceptions governs what happens. This is one reason why the `SVC` exception priority was set to be lower than that of IRQ's.
- When the processor goes to sleep in the Apollo 3 SOC, clocks are turned off and the memory buses no longer function. This means that attached debuggers are not able to read information out of the system. This situation affects the SEGGER RTT used for debug output and is handled by making defining a pre-processor symbol, `USE_SEGGER_RTT`. If this symbol is defined, the main RTT upload buffer is drained before going to sleep.

The key logic used to put the core to sleep and *not* loose the race with an interrupt rests with the fact that the `WFI` instruction ignores the effect of the `PRIMASK` register when deciding if a pending exception will cause the processor to wake back up. The logic works as follows.

1. Set `PRIMASK` to 1. This starts a critical section where no exception with a configurable priority will preempt execution.
2. Check if the background notification queue is still empty. If so, then the race between the background deciding there is no additional work to do and an interrupt coming along and pending an exception has been "won". With `PRIMASK` set to 1, conditions are set to use `WFI` to put the core to sleep and there will be no preemption before the `WFI` instruction is executed[6].
3. After executing `WFI`, the core wakes up if any exception is pending which has a higher execution priority than the current exception (in this case the current exception is the `SVC` exception). This determination ignores the setting of `PRIMASK`, *i.e.* `WFI` ignores the fact that the pending exception may not be made active immediately when the processor wakes back up. This is critical, because between executing the instructions which determine that the background notification queue is empty and executing the `WFI` instruction, an exception can be made pending. Because `PRIMASK` is set to 1, the exception is not made active, but if there is a pending, higher priority exception when `WFI` executes, the execution continues since that is one of the conditions of `WFI` to wake up the processor. This behavior insures that the race between executing the instructions which decide that the conditions for sleeping are appropriate and executing `WFI` does not affect the execution path even if an exception is made pending after the test but before executing `WFI`. Thus there are two races which must be "won" before the core is put to sleep. Arbitrating this situation is like arbitrating between simultaneous interrupt requests at the same priority, in that it can only be handled by innate core logic.
4. If there is a pending, higher priority exception either when `WFI` is executed or anytime thereafter, the core resumes (or simply continues) execution.
5. When execution resumes, `PRIMASK` is still set to 1 and, despite the fact that a higher priority exception is pending, the effects of `PRIMASK` prevents the exception from becoming active and no execution preemption occurs. This allows us to perform any other actions necessary to get the core fully running. On some SOC's this might involve restarting clocks, crystals, or any of a number of SOC specific actions.
6. When all is ready, code can set `PRIMASK` back to 0, which ends the critical section and enables exception preemption. The highest priority pending exception is then made active.

Despite all the convolutions in the logic about putting the core to sleep, the code to implement it is deceptively simple.

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_wait(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(req == SYS_WAIT_REQ) ; (void)req ;
    assert(input == NULL) ; (void)input ;
```

[5] The details of the semantics and recommendations for using `WFI` are contained in section B1.5.19 of the ARMv7-M Archtecture Reference Manual, (ARM DDI 0403E.d ID070218, p. B1-562).

[6] Well, not absolutely none. If things go really wrong, *priority escalation* will force a HardFault which would preempt the execution. But a HardFault is considered to be unrecoverable and the system is destined for a reset operation

```
    assert(output == NULL) ; (void)output;
    assert(error == NULL) ; (void)error ;

    bool wait = true ;
    do {
#               ifdef USE_SEGGER_RTT

            while (SEGGER_RTT_HasDataUp(0) != 0) {          // ❶
                ; // empty
            }

#               endif /* USE_SEGGER_RTT */

        uint32_t mask = stwd_begin_critical_section() ;
    // BEGIN CRITICAL SECTION                                   ❷

        if (bg_queue_is_empty()) {
            __DSB() ;                                       // ❸
            (void)*((uint32_t volatile *)0x5fff0000) ;      // ❹
            __WFI() ;
        } else {
            wait = false ;
        }

        stwd_end_critical_section(mask) ;
    // END CRITICAL SECTION                                     ❺

    } while (wait) ;

    return 0 ;
}
```

❶ Since there is a high probability the core is about to go to sleep and sleeping shuts down the ability of the SEGGER J-Link to communicate with the processor, wait for the RTT buffer on the primary output to drain. The tight loop keeps the processor clocks running while the J-Link has an opportunity to upload the buffered output stream.

❷ At this point, no IRQ's will cause preemption. It is safe to test the state of the background notification queue.

❸ ARM recommends a data synchronization barrier instruction to insure that all data writes have been completed before executing `WFI`. In this particular case, this is not necessary, but at the cost of one instruction, the recommendation is followed.

❹ There is a strange code sequence in the Ambiq HAL code which reads a memory location which is given the symbol, `SYNC_READ` in the code. A comment states that the read "... will hold the bus until all the queued write operations have completed." The address does not appear in the SVD file and there seems to be no mention of it in the reference manual. The implementation in the HAL is rather strangely coded, but the intent seems to be to read the memory location and discard any value. Presumably, the read action has some side effect in the hardware. The memory read is included here because it was found in the Ambiq HAL code.

❺ If an exception caused us to resume after the `WFI` then it will preempt execution at this spot. It is possible for an IRQ hander to run and *not* to place a notification in the background notification queue, *i.e.* an interrupt source may service device internals and *not* need to issue a notification. The core will have awakened anyway. For this reason, the code loops back and tests the queue state again. The loop is terminated only when there is a notification in the queue.

Finally, the implementation of the function which determines if the background notification queue is empty is shown.

```
<<bg req queue: external function declarations>>=
extern bool
bg_queue_is_empty(void) ;
```

The `bg_queue_is_empty` function determines if there is a slot which can be read in the background notification queue. It returns `true` if there is a slot available at the read side of the queue and `false` otherwise.

Implementation

```
<<bg req queue: external function definitions>>=
bool
bg_queue_is_empty(void)
{
    return bip_peek(&sys_bg_notifications, NULL) == NULL ;
}
```

Since the `sys_realm_wait` function sets `PRIMASK` to 1 to disable all exceptions with a configurable priority, consider what happens if a fault occurs during this time. For the specifics here, the question resolves to what happens if during the "peek" at the notification queue a fault occurs. This condition is subject to *priority escalation* to the HardFault priority level (*i.e.* -1). The rules for priority escalation do not apply to this case because:

1. the configurable priority system faults are enabled and all have a priority which is greater than `SVC` exception and
2. the `SVC` instruction is not executed by `bip_peek()`.

Note

The `sys_realm_wait` function is the only place in the system where `PRIMASK` is set to one for the purposes of preventing all preemption and interleaving of instruction execution for the purposes of controlling execution flow. Avoiding the disabling all exceptions helps to maintain the best responsiveness to the system environment and the most protection available from system fault detection. Where there is shared access to data between preemptive and non-preemptive execution, minimizing the scope of the effect is preferred. For example, raising the current execution priority to either inhibit interrupts in general or disabling a particular interrupt narrows the scope of inhibiting preemption. The goal is to minimize the time when all configurable exceptions are inhibited and to let the finer-grained execution prioritization available from the processor architecture provide the arbitration for execution preemption.

Just as for the `sys_realm_get_bg_notification` function, the wait request is assigned an integer encoding and its foreground proxy function is added to the system realm dispatch table.

```
<<sys realm param: constants>>=
#define SYS_WAIT_REQ            2
```

```
<<svc entry: system request functions>>=
[SYS_WAIT_REQ] = sys_realm_wait,
```

Dispatching Notifications into the Background

Background processing must treat notifications coming from foreground with a sense of urgency and dispatch the notifications in a timely manner. After all, notifications indicate conditions to which a reactive system must take action. Timely execution of the notifications resolves to dispatching all the notifications at once until there are no more in the notification queue. The `sys_ctrl_dispatch_notifications` function accomplishes this.

```
<<sys svc req: external declarations>>=
extern int
sys_ctrl_dispatch_notifications(void) ;
```

The `sys_ctrl_dispatch_notifications` function repeatedly obtains the next background notification and executes the proxy function given in the notification until there are no more background notification in the queue. A non-negative return value of the function indicates the number of notifications which were dispatched. A negative return value indicates an error.

Implementation

```
<<sys svc req: external definitions>>=
int
sys_ctrl_dispatch_notifications(void)
{
#   ifndef  BG_NOTIFICATION_MAX_SIZE
#       define  BG_NOTIFICATION_MAX_SIZE    128                          // ❶
#   endif /* BG_NOTIFICATION_MAX_SIZE */

    int notify_count = 0 ;
    char alignas(SVC_DevNotification) req_buf[BG_NOTIFICATION_MAX_SIZE] ;

    int status ;
    for (status = sys_get_bg_notification(sizeof(req_buf), req_buf, NULL) ;
            status > 0 ;
            status = sys_get_bg_notification(sizeof(req_buf), req_buf, NULL)) {
        SVC_DevNotification *notify = (SVC_DevNotification *)req_buf ;
        SVC_DevNotifyProxy proxy = notify->notify_proxy ;
        assert (proxy != NULL) ;

        if (proxy != NULL) {
            proxy(notify) ;
            notify_count++ ;
        }
    }

    return status == 0 ? notify_count : status ;
}
```

❶ A guess as to what the size of the largest notification is likely to be. Generally, it is not advisable to move large quantities of data through the background notification queue. Placing input data into background-supplied buffers is a better strategy for moving larger amounts of data.

Interleaving Background Notifications and Background Processing

All the components needed to design the mechanism for interleaving background notifications with the background processing which responds to the notifications are in place. The design of the `sys_ctrl_dispatch_notifications` function caused *all* background notifications to be dispatched until the Background Notification Queue is empty. For simpler applications (e.g. the examples in this part of the book), the background notification proxy functions can often perform all the computations necessary to respond to the notification. But for larger and more complex applications, it is usually necessary to allow background processing an additional opportunity to complete the processing required by a notification. For example, a notification may start some behavior, but additional computations and perhaps additional notifications are required to complete the response behavior.

To interleave background processing with the dispatch of background notifications, the concept of a background service function is introduced. The purpose of a background service function is to perform a "quantum of work" to resolve the response to background notifications. The definition of *quantum of work* is purposely vague at this time. In Part II of the this book, a detailed definition is presented.

```
<<sys svc req: data type declarations>>=
typedef bool (*BG_ServiceFunc)(void) ;
```

A background service function returns a boolean value to indicate that a work quantum was actually performed. A `true` return value indicates that there could be more work to perform and the background service function need to be invoked again. A `false` return value indicates that the function had no remaining work to perform. The supplied default background service function performs no work. By using a *weak* declaration, it can be overridden for more complex background processing purposes. So, by default, the background notification proxy functions are assumed to perform all the necessary work to respond to a notification. The default background service function is suitable only for simple cases such as the ones used in the current examples.

```
<<sys svc req: external definitions>>=
__attribute__((weak))
bool
sys_ctrl_background_service(void)
{
    return false ;
}
```

Applications which perform background service work, supply their own version of this function. Again, in Part II of this book the background service function which completes the execution model for this design is shown. For now, the default function is sufficient to exercise example and testing code.

The fundamental execution sequencing is then a “busy/wait” loop which:

1. Dispatches all queued background notifications.
2. Invokes the background service function.
3. Checks if there has been a request to exit the execution loop.
4. If the background service indicated that it did not perform any work, then wait for new background notifications.

```
<<sys svc req: external declarations>>=
extern void
sys_ctrl_busy_wait(
    bool volatile *const terminate) ;
```

terminate
A pointer to a boolean variable which is used to determine if the busy/wait loop has synchronized to background execution. Any background execution which sets the value pointed to by `terminate` to `true` causes the loop to exit at the first available opportunity.

```
<<sys svc req: external definitions>>=
void
sys_ctrl_busy_wait(
    bool volatile *const terminate)
{
    assert(terminate != NULL) ;

    *terminate = false ;
    for (;;) {
        int notify_status = sys_ctrl_dispatch_notifications() ;      // ❶
        rtcheck(notify_status >= 0) ;

        bool performed_service = sys_ctrl_background_service() ;

        if (*terminate) {                                            // ❷
            break ;
```

```
        }

        if (!performed_service) {
            sys_wait() ;                                                    // ❸
        }
    }
}
```

❶ After invoking `sys_ctrl_dispatch_notifications`, the Background Notification Queue is empty.

❷ It is important to check for termination after all notifications have been dispatched and the system background has been serviced. These are the two actions which cause code to run which could modify the value of the variable pointed to by `terminate`.

❸ If dispatching of the background notifications did not result in the background service performing any work, then there is no remaining work to do, so it is time to wait. Note there is a race between deciding there is nothing left to do and an IRQ Handler posting a background notification (interrupts are enabled here). If an IRQ hander did "win the race" and post a background notification, `sys_wait` returns immediately.

There are a couple of noteworthy implications about the `sys_ctrl_busy_wait` function:

- The execution loop exits if background processing writes to the variable pointed to by `terminate`. This allows for graceful termination of the loop. When `sys_ctrl_busy_wait` is used as the "main" loop for an application, it is typically invoked from `main` and if it returns, the application has an opportunity to execute clean up code before returning from `main` and thus terminating the program. This is also useful for test code where it is necessary to enter the execution loop but it is also necessary to regain control over the execution flow to determine the status of a test.
- It is possible to invoke `sys_ctrl_busy_wait` recursively, *i.e.* it is possible for background processing to "busy wait" in place and contrive to use the termination variable to synchronize, eventually, to a particular linear flow. This usage is subject to abuse and should generally be avoided ,*i.e.* there should be only a single invocation of `sys_ctrl_busy_wait` in a program. But when a particular interactive situation arises where processing must wait for some notification and then proceed in sequence, it is an available tool.

A use case for simple situations is to use the background notification closure data as a pointer to a termination variable and then have the background notification proxy use the closure data pointer to cause the synchronization. The following background proxy function implements this simple case.

```
<<dev svc req: external declarations>>=
extern void
svc_proxy_var_sync(
    SVC_DevNotification const *const notification) ;
```

notification
A pointer to the notification information for a background notification request.

The `svc_proxy_var_sync` function is a simple variable synchronization proxy which assumes the closure data in `notification` is a pointer to a boolean variable. When dispatched, the function sets the variable pointed to by the closure data to `true`.

Implementation

```
<<dev svc req: external function definitions>>=
void
svc_proxy_var_sync(
    SVC_DevNotification const *const notification)
{
    assert(notification != NULL) ;
```

```
    bool volatile *const sync_var_ref =
            (bool volatile *const)notification->notify_closure ;
    rtcheck(sync_var_ref != NULL) ;

    *sync_var_ref = true ;
}
```

Code Layout

The layout of code files for this chapter is more complicated than seen previously. In addition to the typical file split in "C" between header and code, there is an additional division between foreground and background. In a subsequent chapter, the foreground/background split is used to control the placement in memory that enforces memory usage constraints between foreground and background execution. Since foreground and background requests and proxies must agree on data types used as arguments, additional header files are used to share that information.

The arrangement of this section starts with the files needed to operate the SVC exception mechanism and the dispatch of foreground proxies for the two service realms. The system realm is presented next, followed by the device realm code for the watchdog timer. Finally, there is some common code that is used in multiple places and which has been extracted into separate files.

SVC Interface

The files in this section contain the declaration and definitions associated with the SVC interface. This includes the two realm handlers and the SVC exception handler.

svc_req.h

```
<<svc_req.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Background service requests made through the SVC mechanism.
 *--
 */
#ifndef SVC_REQ_H_
#define SVC_REQ_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include "sys_realm_param.h"
#include "dev_realm_param.h"
/*
 * External Declarations
 */
<<svc req: external declarations>>

#endif /* SVC_REQ_H_ */
```

svc_req.c

```
<<svc_req.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for SVC based background requests.
 *--
 */
/*
 * Include Files
 */
#include "sys_realm_param.h"
#include "dev_realm_param.h"
/*
 * External Function Definitions
 */
<<svc req: external function definitions>>
```

svc_handler.c

```
<<svc_handler.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation of SVC exception handler and first level proxy function
 *   dispatch.
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "useful.h"
#include "rtcheck.h"
#include "svc_req_errors.h"
#include "sys_realm_proxy.h"
#include "dev_realm_proxy.h"
<<svc handler: device include files>>
/*
 * Static Function Definitions
 */
<<svc handler: static function definitions>>
/*
 * External Function Definitions
 */
<<svc handler: external function definitions>>
```

System Realm Service Files

The system realm services are contained in five files:

1. `sys_svc_req.h` contains the function prototypes for system realm services that can be invoked by background processing.
2. `sys_svc_req.c` contains the function definitions for background portion of the system realm request.
3. `sys_realm_param.h` file contains the data structure definitions for the interfaces to the foreground proxy functions which implement the system realm service requests. Both sides of `SVC` exception execution must agree on the structure of the data passed through the pointer values which are the arguments. Since the `SVC` exception interface passes arguments through without any interpretation, the declarations in this file are needed in two places:
    a. The background side implementation of the service request. The request parameters must be marshaled into the data structures required by the foreground proxy.
    b. The foreground proxy implementation needs to access the requests parameters as part of its implementation of the request.
4. `sys_realm_proxy.h` contains the function prototypes for the system realm foreground proxy functions.
5. `sys_realm_proxy.c` contains the function definitions for the system realm proxy functions that are associated with the background service requests.

sys_svc_req.h

```
<<sys_svc_req.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Function prototypes for system service requests.
 *--
 */
#ifndef SYS_SVC_REQ_H_
#define SYS_SVC_REQ_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdnoreturn.h>
#include "sys_realm_param.h"
/*
 * Data Type Declarations
 */
<<sys svc req: data type declarations>>
/*
 * External Declarations
 */
<<sys svc req: external declarations>>

#endif /* SYS_SVC_REQ_H_ */
```

sys_svc_req.c

```
<<sys_svc_req.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   System background service requests implementation.
 *--
 */

/*
 * Include files
 */
#include <stdlib.h>
#include <stdalign.h>
#include <stdarg.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
#include "rtcheck.h"
#include "svc_req.h"
#include "sys_svc_req.h"
#include "dev_svc_req.h"
#include "svc_req_errors.h"
#include "sys_realm_proxy.h"
#include "panic.h"
/*
 * External Functions
 */
<<sys svc req: external definitions>>
```

sys_realm_param.h

```
<<sys_realm_param.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Parameter interface data structures for system realm requests.
 *--
 */
#ifndef SYS_REALM_PARAM_H_
#define SYS_REALM_PARAM_H_

/*
 * Include Files
 */
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
```

```
#include "svc_param.h"
<<sys realm param: include files>>
/*
 * Constants
 */
<<sys realm param: constants>>
/*
 * Data Type Declarations
 */
<<sys realm param: data type declarations>>

#endif /* SYS_REALM_PARAM_H_ */
```

sys_realm_proxy.h

```
<<sys_realm_proxy.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Interfaces for system realm proxy functions.
 *--
 */
#ifndef SYS_REALM_PROXY_H_
#define SYS_REALM_PROXY_H_

/*
 * Include files
 */
#include "sys_realm_param.h"
/*
 * Data Type Declarations
 */
<<sys realm proxy: interface data type declarations>>
/*
 * External Declarations
 */
<<sys realm proxy: external declarations>>

#endif /* SYS_REALM_PROXY_H_ */
```

sys_realm_proxy.c

```
<<sys_realm_proxy.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for system realm foreground proxy functions.
 *--
```

```
 */

/*
 * Include files
 */
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <inttypes.h>
#include "svc_req_errors.h"
#include "svc_proxy.h"
#include "sys_realm_param.h"
#include "sys_realm_proxy.h"
#include "bg_req_queue.h"
#include "useful.h"
#include "sys_twiddle.h"
#include "dev_twiddle.h"
#include "panic.h"
#include "sys_svc_req.h"
#include "rtcheck.h"
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
#ifdef  USE_UART
#   include "dev_realm_uart_proxy.h"
#endif /*  USE_UART */
/*
 * Static Data Definitions
 */
<<sys realm proxy: static data definitions>>
/*
 * Static Function Definitions
 */
<<sys realm proxy: static function definitions>>
/*
 * External Function Definitions
 */
<<sys realm proxy: external function definitions>>
```

Device Realm Service Files

The code files for device realm services follow a similar pattern as those for system realm services. The device realm has an additional level of dispatch. This causes us to add a header file to hold declarations that apply to all device realm requests.

The interfacing and implementation files are separate for each device. Each logic device uses five files for its interface and implementation. In this chapter, the watchdog timer was used as an example of device realm services. Those files are given here and establish the pattern to be used in later chapters for other device control code. The five watchdog device files are:

1. `dev_svc_wdog_req.h`
2. `dev_svc_wdog_req.c`
3. `dev_realm_wdog_param.h`
4. `dev_realm_wdog_proxy.h`
5. `dev_realm_wdog_proxy.c`

dev_realm_param.h

The dev_realm_param.h file contains the data structure definitions for the interfaces to the device realm services. This file is the counterpart to sys_realm_param.h Because the device realm is further partitioned for the individual devices, this header file contains only the common definitions shared among all the distinct devices.

```
<<dev_realm_param.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Common parameter interface data structures for device realm requests.
 *--
 */
#ifndef DEV_REALM_PARAM_H_
#define DEV_REALM_PARAM_H_
/*
 * Include Files
 */
#include <stddef.h>
#include <stdint.h>
#include "dev_svc_req.h"
#include "svc_param.h"
#include "bit_twiddle.h"
/*
 * Constants
 */
<<dev realm param: constants>>
/*
 * Data Type Declarations
 */
<<dev realm param: data type declarations>>
/*
 * Static Inline Definitions
 */
<<dev realm param: static inline definitions>>

#endif /* DEV_REALM_PARAM_H_ */
```

dev_realm_proxy.h

The dev_realm_proxy.h file contains common definitions used by all the device realm foreground proxy functions.

```
<<dev_realm_proxy.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Interfaces for device realm proxy functions.
 *--
 */
#ifndef DEV_REALM_PROXY_H_
```

```
#define DEV_REALM_PROXY_H_

/*
 * Include files
 */
#include "dev_svc_req.h"
#include "dev_realm_param.h"
#include "rtcheck.h"
/*
 * Static Inline Definitions
 */
<<dev realm proxy: static inline definitions>>

#endif /* DEV_REALM_PROXY_H_ */
```

dev_svc_wdog_req.h

```
<<dev_svc_wdog_req.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Function prototypes for watchdog service requests.
 *--
 */
#ifndef DEV_SVC_WDOG_REQ_H_
#define DEV_SVC_WDOG_REQ_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "dev_svc_req.h"
#include "dev_realm_wdog_param.h"
/*
 * External Declarations
 */
<<dev svc wdog req: external declarations>>

#endif /* DEV_SVC_WDOG_REQ_H_ */
```

dev_svc_wdog_req.c

```
<<dev_svc_wdog_req.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
```

```
 *   Device Realm Watchdog Request Implementation
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "useful.h"
#include "svc_req_errors.h"
#include "svc_req.h"
#include "dev_svc_req.h"
#include "dev_svc_wdog_req.h"
#include "dev_realm_wdog_param.h"
/*
 * External Functions
 */
<<dev svc wdog req: external definitions>>
```

dev_realm_wdog_param.h

```
<<dev_realm_wdog_param.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Parameter interface data structures for watchdog device requests.
 *--
 */
#ifndef DEV_REALM_WDOG_PARAM_H_
#define DEV_REALM_WDOG_PARAM_H_
/*
 * Include Files
 */
#include "dev_realm_param.h"
/*
 * Constants
 */
<<dev realm wdog param: constants>>
/*
 * Data Type Declarations
 */
<<dev realm wdog param: data type declarations>>

#endif /* DEV_REALM_WDOG_PARAM_H_ */
```

dev_realm_wdog_proxy.h

```
<<dev_realm_wdog_proxy.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
```

```
 *   Code to Models
 *
 * Module:
 *   Interfaces for device realm wdog proxy functions.
 *--
 */
#ifndef DEV_REALM_WDOG_PROXY_H_
#define DEV_REALM_WDOG_PROXY_H_

/*
 * Include files
 */
#include "dev_realm_param.h"
/*
 * External Declarations
 */
<<dev realm wdog proxy: external declarations>>

#endif /* DEV_REALM_WDOG_PROXY_H_ */
```

dev_realm_wdog_proxy.c

```
<<dev_realm_wdog_proxy.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for device realm watchdog proxy functions.
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "svc_req_errors.h"
#include "svc_proxy.h"
#include "dev_realm_proxy.h"
#include "dev_realm_wdog_param.h"
#include "bg_req_queue.h"
#include "rtcheck.h"
#include "panic.h"
#include "apollo3.h"
#include "bit_twiddle.h"
#include "sys_twiddle.h"
/*
 * Static Data Definitions
 */
<<dev realm wdog proxy: static data definitions>>
/*
 * Static Function Definitions
 */
<<dev realm wdog proxy: static function definitions>>
/*
 * External Function Definitions
 */
```

```
<<dev realm wdog proxy: external function definitions>>
```

SVC Device Class Handler Interface

Each logical device provides a class level dispatch function as part of the two level proxy dispatch used in the device realm. In this section, the watchdog device was presented and so the prototype for its class level dispatch function is required to be included in the device realm `SVC` handler. In subsequent chapters as new device control code is presented, the header file for that proxy code must also be included in the list of handler include files.

```
<<svc handler: device include files>>=
#include "dev_realm_wdog_proxy.h"
```

Common Background Service Requests

There are several files which are common to both the system and device realm request code.

svc_req_errors.h

The `svc_req_errors.h` contains the error number encoding for the background request functions. The return value of all background request functions will be a negative value of one of these constants.

```
<<svc_req_errors.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Error number encoding for SVC requests.
 *--
 */
#ifndef SVC_REQ_ERRORS_H_
#define SVC_REQ_ERRORS_H_

/*
 * Constants
 */
<<svc req errors: constants>>
/*
 * External Function Declarations
 */
<<svc req errors: external function declarations>>

#endif /* SVC_REQ_ERRORS_H_ */
```

svc_req_errors.c

```
<<svc_req_errors.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
```

```
 *   Code to Models
 *
 * Module:
 *   Functions for error numbers
 *--
 */

/*
 * Include files
 */
#include <stdlib.h>
#include "svc_req_errors.h"
#include "useful.h"
/*
 * External Function Definitions
 */
<<svc req errors: external function definitions>>
```

svc_param.h

The `svc_param.h` file contains common definitions for SVC request parameters.

```
<<svc_param.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Generic parameter interface data structures and operations for
 *   service requests.
 *--
 */
#ifndef SVC_PARAM_H_
#define SVC_PARAM_H_

/*
 * Include Files
 */
#include <stddef.h>
#include <stdint.h>
/*
 * Macros
 */
<<svc param: macros>>
/*
 * Data Type Declarations
 */
<<svc param: data type declarations>>

#endif /* SVC_PARAM_H_ */
```

dev_svc_req.h

The `dev_svc_req.h` header file contains data structure definitions for device notifications and the definitions for the encoding of the device request parameters.

```
<<dev_svc_req.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Function prototypes for device service requests.
 *--
 */
#ifndef DEV_SVC_REQ_H_
#define DEV_SVC_REQ_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
/*
 * Data Type Declarations
 */
<<dev svc req: data type declarations>>
/*
 * External Declarations
 */
<<dev svc req: external declarations>>

#endif /* DEV_SVC_REQ_H_ */
```

dev_svc_req.c

```
<<dev_svc_req.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for common device service request.
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include <string.h>
#include <stdbool.h>
#include "svc_req_errors.h"
#include "dev_svc_req.h"
#include "rtcheck.h"
/*
 * External Function Definitions
 */
<<dev svc req: external function definitions>>
```

Common Foreground Proxy Files

svc_proxy.h

```
<<svc_proxy.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Common code for foreground proxy functions.
 *--
 */
#ifndef SVC_PROXY_H_
#define SVC_PROXY_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include "svc_param.h"
/*
 * Macros
 */
<<svc proxy: macros>>
/*
 * Static Inline Functions
 */
<<svc proxy: static inline functions>>
/*
 * External Function Declarations
 */
<<svc proxy: external function declarations>>

#endif /* SVC_PROXY_H_ */
```

```
<<svc_proxy.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for common service request proxy functions.
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include <string.h>
#include <stdbool.h>
#include "svc_req_errors.h"
#include "svc_proxy.h"
```

```
#include "rtcheck.h"
/*
 * External Function Definitions
 */
<<svc proxy: external function definitions>>
```

bg_req_queue.h

The `bg_req_queue.h` file contains the interface definitions used by the foreground proxy code which accesses the notification queue.

```
<<bg_req_queue.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Background request queue operations interface
 *--
 */
#ifndef BG_REQ_QUEUE_H_
#define BG_REQ_QUEUE_H_
/*
 * Include Files
 */
#include <stddef.h>
#include <stdbool.h>
#include "dev_svc_req.h"
/*
 * External Function Declarations
 */
<<bg req queue: external function declarations>>

#endif /* BG_REQ_QUEUE_H_ */
```

bg_req_queue.c

```
<<bg_req_queue.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Background request queue operations implementation
 *--
 */

/*
 * Include Files
 */
#include <string.h>
#include "bg_req_queue.h"
```

```
#include "svc_proxy.h"
#include "svc_req_errors.h"
#include "bip_buffer.h"
#include "exc_priority.h"
#include "sys_twiddle.h"
#include "rtcheck.h"
/*
 * Constants
 */
<<bg req queue: constants>>
/*
 * Static Data Definitions
 */
<<bg req queue: static data definitions>>
/*
 * External Function Definitions
 */
<<bg req queue: external function definitions>>
```

Exception Priorities

exc_priority.h

```
<<exc_priority.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Definitions of the priorities given to configurable exceptions.
 *--
 */
#ifndef EXC_PRIORITY_H_
#define EXC_PRIORITY_H_

/*
 * Constants
 */
<<exc priority: constants>>

#endif /* EXC_PRIORITY_H_ */
```

system_exc_priority_init.h

```
<<system_exc_priority_init.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   System exception priority initialization.
 *--
```

```
 */

/*
 * Include Files
 */
#include <stdint.h>
#include <stdbool.h>
#include "apollo3.h"
#include "system_apollo3.h"
#include "sys_twiddle.h"
#include "dev_twiddle.h"
#include "exc_priority.h"

/*
 * External Function Definitions
 */
<<system exc priority init: external function definitions>>
```

Example

The following example shows all the concepts in this part of the book in action. The program initializes the watchdog timer and starts it. As the watchdog IRQ posts notifications, they are dispatched and the system is put to sleep waiting for the next notification.

```
<<crossing-test: external functions>>=
int
main(
    int argc,
    char **argv)
{
#       ifdef USE_SEGGER_RTT
    SEGGER_RTT_Init() ;
#       endif /* USE_SEGGER_RTT */

    char time_buf[32] ;

    (void)sys_util_eptime_ticks_format(NULL, sizeof(time_buf), time_buf) ;
    printf("timestamp = %s\n", time_buf) ;

    dev_wdog_init(0, dev_util_wdog_ms_to_ticks(4000), dev_util_wdog_ms_to_ticks(3500),
            wdog_notify_function, 0) ;

#       ifdef RELEASE_BUILD                                    /* ❶ */
    dev_wdog_start(0, true) ;
#       else  /* RELEASE_BUILD */
    dev_wdog_start(0, false) ;
#       endif /* RELEASE_BUILD */

    bool volatile forever ;                                    // ❷
    sys_ctrl_busy_wait(&forever) ;

    return 0 ;
}
```

❶ Recall that the Apollo 3 watchdog timer cannot be set to halt when the debugger is active. Debug builds never cause the system to reset from lack of watchdog service, but breakpoints do allow resumption. It is necessary to run the release build to insure that the watchdog is actually monitoring something.

❷ Since `forever` is an automatic variable and there is no watchdog notification proxy which can set its value to true, the call to `sys_ctrl_busy_wait` does not return.

The background proxy function for the watchdog performs the restart of the watchdog timer along with printing some information.

```
<<crossing-test: static functions>>=
static void
wdog_notify_function(
    SVC_DevNotification const *params)
{
    dev_wdog_restart(params->device_instance) ;

    char time_buf[32] ;

    (void)sys_util_eptime_ticks_format(NULL, sizeof(time_buf), time_buf) ;
    printf("WDOG: notify: %s\n", time_buf) ;
}
```

```
<<crossing-the-divide-test.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Farfalle
 *
 * Module:
 *   Tests for system call interface.
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <inttypes.h>
#include "sys_svc_req.h"
#include "dev_svc_wdog_req.h"
#include "bit_twiddle.h"
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
/*
 * Constants
 */
<<crossing-test: constants>>
/*
 * Data Type Declarations
 */
<<crossing-test: data type declarations>>
/*
 * External Declarations
 */
<<crossing-test: external declarations>>
/*
 * Forward References
 */
```

```
<<crossing-test: forward references>>
/*
 * Static Inline Functions
 */
<<crossing-test: static inline functions>>
/*
 * External Inline Functions
 */
<<crossing-test: external inline functions>>
/*
 * Static Data
 */
<<crossing-test: static data>>
/*
 * Static Functions
 */
<<crossing-test: static functions>>
/*
 * External Functions
 */
<<crossing-test: external functions>>
```

Chapter 5

Shoring Up the Foundation

To recap, the previous chapters accomplished:

a. Start up code is in place that performs the necessary initializations to reach the `main()` function.

b. SEGGER RTT has been integrated into the I/O functions from `newlib` to enable formatted output.

c. Code execution has been separated into privileged and unprivileged modes.

d. A mechanism based upon the `SVC` exception was built to allow unprivileged and privileged code to interact.

However, this foundation is not strong enough to support robust deployment. The handling of system faults and missing IRQ handlers does not support saving any information which might help in diagnosing the fault. This chapter shores up the exception and error handling for a wider set of conditions.

Software detected faults

The processor core generates system faults when it detects that the external environment is not responding properly or when the executing program does not operate the processor properly. For example, misbehaving memory can result in a Bus Fault and software execution of a divide operation with a zero value for the divisor can result in a Usage Fault.

Similar circumstances can arise when operating peripheral devices or manipulating system software structures. For example, most UART devices offer status and an interrupt, to indicate conditions such as transmitter overflow or receiver underflow. Transmitter overflow results from attempting to place data into the transmitter when it is already full. Similarly, receiver underflow occurs when software attempts to read the receiver when is empty. Both of these conditions indicate that the UART is being operated improperly since there is always sufficient status information available from the UART registers to determine if the write or read operation is allowed. There are many circumstances where hardware peripherals can detect improper operation by software. These situation often have interrupts associated with them that can be used to detect run time operational problems.

A **panic condition** is defined by analogy to a system fault. The panic condition is not intended to recoverable. It signifies a situation where software has detected a condition and does not have any way to proceed without the possibility of uncontrolled execution.

An easy solution to a panic would be to invoke SystemAbend(). Recall that invoking `SystemAbend` was defined as "hitting rock bottom". But, just invoking `SystemAbend()` does not give any information about the circumstances of the detected problem. Some clue of the exceptional condition which caused the panic condition needs to be saved.

You must also consider whether the panic condition was detected by privileged or unprivileged code. Only privileged code can invoke `SystemAbend()` directly. Storing away the circumstances of the panic in memory that survives reset is also a privileged operation. However, there are situations that can occur during unprivileged execution that constitute a panic condition and action needs to be taken.

In this section, two flavors of functions that declare a panic condition are presented.

1. The `panic()` function is suitable for invoking from privileged foreground processing.
2. The `sys_panic()` function, which is a background request to execute the `panic` function, is provided for use by background processing.

Both functions have the same interface which is patterned after `printf`. Errors indicated by invoking `panic` or `sys_panic` are considered unrecoverable. Both functions ultimately invoke `SystemAbend`. Recall that the provided `SystemAbend` function is a *weak* symbol, whose default implementation is to reset the MCU. Projects may provide a replacement to handle any special system circumstances. There is no return and ultimately the system is reset. It is not a mechanism to be used casually and the expectation is that code implementing application logic does *not* invoke `sys_panic` directly.

Panic

The `panic` function has an interface like `printf` so that the caller can pass along data which is formatted before being stored in a memory location which survives reset.

```
<<panic: external declarations>>=
extern noreturn void
panic(
    char const *fmt,
    ...) ;
```

fmt
: A `printf` style format string.

...
: A variable number of arguments as required by the `fmt` string.

The `panic` function stores a formatted error message in a memory location which survives system reset before causing an abnormal system termination. This function can only be invoked from privileged execution. Unprivileged execution must invoke `sys_panic` to achieve the same result.

Standard definitions to handle the `noreturn` and the macros used to access variable length argument lists are obtained from the usual header files.

```
<<panic: interface include files>>=
#include <stddef.h>
#include <stdarg.h>
#include <stdnoreturn.h>
#include "useful.h"
#include "rtcheck.h"
```

Implementation

```
<<panic: external functions>>=
__attribute__((format(printf, 1, 2)))                                   // ❶
noreturn void
panic(
    char const *fmt,
    ...)
{
    size_t remain ;
    char *msg = panic_buf_alloc(&remain) ;                              // ❷

    uint64_t ts = dtwd_get_timestamp() ;
    int ts_len = sys_util_eptime_ticks_format(&ts, remain, msg) ;       // ❸

    if (ts_len < remain) {
```

```
        remain -= ts_len ;
        char *trailing = msg + ts_len ;

        va_list ap ;
        va_start(ap, fmt) ;

        vsnprintf(trailing, remain, fmt, ap) ;

        va_end(ap) ;
    }

#       if defined USE_SEGGER_RTT
    if (stwd_debug_enabled() || stwd_debugmon_enabled()) {
        printf("%s\n", msg) ;

        while (SEGGER_RTT_HasDataUp(0) != 0) {                          // ❹
            ; // empty
        }
    }
#       elif defined USE_UART
    (void)puts_priv(msg) ;
#       endif /* USE_SEGGER_RTT */

    SystemAbend() ;
}
```

❶ This incantation informs GCC that `panic` takes a `printf` format string and it should verify the arguments against the expectations of the format string.

❷ The message associated with `panic` is stored in a specially allocated buffer. See below.

❸ This function formats a timestamp in ticks to a readable string. It is shown in a subsequent chapter.

❹ Since this execution path is to break into the debugger, make sure to output the panic output.

For the implementation, macros for variable length arguments and some string / memory functions are required.

```
<<panic: include files>>=
#include <string.h>
#include "panic.h"
#include "sys_svc_req.h"
```

The output may be routed to the SEGGER RTT or the UART. UART device code is presented in a subsequent chapter.

```
<<panic: include files>>=
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
#ifdef  USE_UART
#   include "dev_realm_uart_proxy.h"
#endif /*  USE_UART */
```

`Panic` also uses timestamps and makes a call to `SystemAbend()`.

```
<<panic: include files>>=
#include "dev_twiddle.h"
#include "sys_twiddle.h"
#include "system_apollo3.h"
```

Sys_panic

The `sys_panic` function has the same interface as `panic`, but may be invoked from unprivileged code. As expected, the function formulates a request to its corresponding foreground proxy.

```
<<sys svc req: external declarations>>=
extern noreturn void
sys_panic(
    char const *fmt,
    ...) ;
```

fmt
: A `printf` style format string.

...
: A variable number of arguments as required by the `fmt` string.

The `sys_panic` function stores a formatted error message in a recoverable location before causing an abnormal system termination. This function can only be invoked from privileged execution. Unprivileged execution must invoke `sys_panic` to achieve the same result.

The implementation of `sys_panic` follows the pattern established in a previous chapter of the book. The `sys_panic` function makes a request and there is a foreground proxy which fulfills the request.

One aspect of the implementation for `sys_panic` is different from other system calls seen so far. The `SVC` interface does not support variadic function arguments. Since the variadic arguments are concerned with text formatting, the text is formatted in the background and the resultant string is passed as an input parameter to the foreground proxy function. Consequently, a request input structure must be defined.

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysPanicRequestInput,
    char const *const msg ;
) ;
```

Implementation

```
<<sys svc req: external definitions>>=
__attribute__((format(printf, 1, 2)))
noreturn void
sys_panic(
    char const *fmt,
    ...)
{
    char msg[PANIC_BUF_SIZE] ;

    va_list ap ;
    va_start(ap, fmt) ;
    (void)vsnprintf(msg, sizeof(msg), fmt, ap) ;                    // ❶
    va_end(ap) ;

    SVC_SysPanicRequestInput req_input = {
        .block_size = sizeof(SVC_SysPanicRequestInput),
        .msg = msg,
    } ;

    (void)sys_realm_svc_call(SYS_PANIC, &req_input, NULL, NULL) ;

    abort() ;                                                       // ❷
}
```

❶ Note that formatted output which exceeds the size of the message buffer is silently truncated.

❷ The invocation of `abort()` is present to quiet a compiler warning for a, "noreturn function does return", warning. Here, it is "known" that the foreground proxy will not return even if `sys_realm_svc_call` does return in every other case. However, the compiler can't be told of this special circumstance. As a workaround, a function which the compiler understands will not return is placed in the execution flow despite there being no expectation it is never invoked. All this is to quiet a compiler warning that would otherwise require examination each time the source is built.

As before, the proxy function is given an unique identifying number and inserted it into the jump table used for dispatching in the system realm.

```
<<sys realm param: constants>>=
#define SYS_PANIC     3
```

```
<<svc entry: system request functions>>=
[SYS_PANIC] = sys_realm_panic,
```

The foreground proxy function which implements the operation to obtain the next background service request has the following prototype:

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_panic(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_panic(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(req == SYS_PANIC) ; (void)req ;

    SVC_SysPanicRequestInput req_input ;
    int result = copy_in_svc_param(input, sizeof(req_input),
        (SVC_RequestParam *const)&req_input) ;

    rtcheck_zero_return(result, result) ;
    rtcheck_return(req_input.msg != NULL, -ERR_INVALID_PARAM) ;

    size_t buf_size ;
    char *msg = panic_buf_alloc(&buf_size) ;

    uint64_t ts = dtwd_get_timestamp() ;                                    // ❶
    int ts_len = sys_util_eptime_ticks_format(&ts, buf_size, msg) ;

    if (ts_len < buf_size) {
        buf_size -= ts_len ;
        char *tail = msg + ts_len ;

        size_t msg_len = strlen(req_input.msg) ;
        size_t copy_count = (buf_size <= msg_len) ?
                buf_size - 1 : msg_len ;                                    // ❷
        memcpy_from_unpriv(tail, req_input.msg, copy_count) ;               // ❸
        tail[copy_count] = '\0' ;
    }
```

```
#       if defined USE_SEGGER_RTT
    if (stwd_debug_enabled() || stwd_debugmon_enabled()) {
        printf("%s\n", msg) ;

        while (SEGGER_RTT_HasDataUp(0) != 0) {
            ; // empty
        }
    }

#       elif defined USE_UART
    (void)puts_priv(msg) ;                                          // ❹
#       endif /* USE_SEGGER_RTT */

    SystemAbend() ;

    return result ;     // Make the compiler happier about the abuse of
                        // the function prototype.
}
```

❶ The panic message from the background is timestamped by the foreground proxy.

❷ If the number of bytes remaining in the panic buffer is equal to the number of characters in the message, then there is no place to store the `NUL` character. A characters is truncated from the message to insure the string is `NUL` terminated.

❸ You can't just add the panic message as a `%s` parameter to `snprintf` as that implies a copy from unprivileged memory using privileged execution.

❹ A UART function for privileged output.

Panic message storage

Storage of panic message is allocated in the `.noinit_data` and `.noinit_bss` sections. First, sizing information must be specified. The values chosen here are reasonable for expected use.

```
<<panic: constants>>=
#ifndef PANIC_BUF_SIZE
#   define PANIC_BUF_SIZE       256U
#endif /* PANIC_BUF_SIZE */
```

A 256 byte buffer should be sufficient for "a long line" of text.

```
<<panic: constants>>=
#ifndef PANIC_BUF_COUNT
#   define PANIC_BUF_COUNT       4U
#endif /* PANIC_BUF_COUNT */
```

Space for saving up to four panic messages is provided. Total storage pool size is then only 1KiB.

```
<<panic: static data>>=
__attribute__((section(".noinit_bss")))
static char panic_buf_pool[PANIC_BUF_SIZE * PANIC_BUF_COUNT] ;
```

A pointer is needed to keep track of the next place in the storage pool memory.

```
<<panic: static data>>=
__attribute__((section(".noinit_data")))
static char *panic_buf_allocator = panic_buf_pool ;
```

This design allocates panic message buffers in the simplest fashion. The allocation structure is a simple circular queue that does *not* account for overflow. Simply the last `PANIC_BUF_COUNT` number of messages are preserved.

```
<<panic: external declarations>>=
char *
panic_buf_alloc(
    size_t *buf_size_ref) ;
```

```
<<panic: external functions>>=
char *
panic_buf_alloc(
    size_t *buf_size_ref)
{
    char *buf = panic_buf_allocator ;
    memset(buf, 0, PANIC_BUF_SIZE) ;

    panic_buf_allocator += PANIC_BUF_SIZE ;
    if (panic_buf_allocator >= (panic_buf_pool + COUNTOF(panic_buf_pool))) {
        panic_buf_allocator = panic_buf_pool ;
    }

    if (buf_size_ref != NULL) {
        *buf_size_ref = PANIC_BUF_SIZE ;
    }

    return buf ;
}
```

```
<<panic: external declarations>>=
void
panic_buf_clear(void) ;
```

```
<<panic: external functions>>=
void
panic_buf_clear(void)
{
    memset(panic_buf_pool, 0, sizeof(panic_buf_pool)) ;
    panic_buf_allocator = panic_buf_pool ;
}
```

```
<<panic: external declarations>>=
char const *
panic_buf_peek(
    size_t which) ;
```

```
<<panic: external functions>>=
char const *
panic_buf_peek(
    size_t which)
{
    rtcheck_max_return(which, PANIC_BUF_COUNT, NULL) ;

    return &panic_buf_pool[which * PANIC_BUF_SIZE] ;
}
```

Panic History Requests

This section presents a set of background requests to operate on the panic message store. Two operations are provided:

a. Get a panic message from the storage pool.

b. Clear the storage pool.

Get a Panic Message

```
<<sys svc req: external declarations>>=
extern int
sys_panic_msg_get(
    size_t which,
    size_t buf_size,
    char buf[buf_size]) ;
```

which
A selector value for which panic message is to be fetched. Values must range from 0 to PANIC_BUF_COUNT.

buf_size
The number of bytes pointed to by `buf`.

buf
A pointer to a memory object where the panic message is placed.

The `sys_panic_msg_get` function retrieves the panic message indicated by `which` and copies it into the buffer pointed to by `buf`. No more than `buf_size` bytes are copied. The message returned in `buf` is always a `NUL` terminated string. The return value is zero upon success and negative otherwise.

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysPanicMsgGetInput,
    size_t which ;
    size_t buf_size ;
    char *buf ;
) ;
```

Implementation

```
<<sys svc req: external definitions>>=
int
sys_panic_msg_get(
    size_t which,
    size_t buf_size,
    char buf[buf_size])
{
    SVC_SysPanicMsgGetInput input = {
        .block_size = sizeof(SVC_SysPanicMsgGetInput),
        .which = which,
        .buf_size = buf_size,
        .buf = buf,
    } ;

    return sys_realm_svc_call(SYS_PANIC_MSG_GET, &input, NULL, NULL) ;
}
```

As usual, a request encoding must be supplied and the pointer to the foreground proxy is placed in the system realm dispatch table.

```
<<sys realm param: constants>>=
#define SYS_PANIC_MSG_GET       9
```

```
<<svc entry: system request functions>>=
[SYS_PANIC_MSG_GET] = sys_realm_panic_msg_get,
```

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_panic_msg_get(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

Implementation

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_panic_msg_get(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(req == SYS_PANIC_MSG_GET) ; (void)req ;
    assert(output == NULL) ; (void)output;
    assert(error == NULL) ; (void)error ;

    SVC_SysPanicMsgGetInput msg_input ;
    int status = copy_in_svc_param(input, sizeof(msg_input), &msg_input) ;
    rtcheck_zero_return(status, status) ;
    rtcheck_not_NULL_return(msg_input.buf, -ERR_INVALID_PARAM) ;

    char const *msg = panic_buf_peek(msg_input.which) ;
    rtcheck_not_NULL_return(msg, -ERR_OPERATION_FAILED) ;
    size_t copy_size = msg_input.buf_size < PANIC_BUF_SIZE ?
            msg_input.buf_size : PANIC_BUF_SIZE ;
    copy_size -= 1 ;                                                    // ❶

    memcpy_to_unpriv(msg_input.buf, msg, copy_size) ;
    WRITE_UNPRIV('\0', (uint8_t *)&msg_input.buf[copy_size - 1]) ;

    return 0 ;
}
```

❶ Space for the `NUL` terminator.

Clear Panic Messages

In addition to obtaining the panic messages, a system service is provided to clear out the panic message block. The interface requires no parameters and the implementation follows the established pattern. The system service does nothing more than providing an unprivileged interface to the `panic_buf_clear` function.

```
<<sys svc req: external declarations>>=
extern int
sys_panic_msg_clear(void) ;
```

The `sys_panic_msg_clear` function deletes all panic messages from the panic message storage pool.

Implementation

```
<<sys svc req: external definitions>>=
int
sys_panic_msg_clear(void)
{
    return sys_realm_svc_call(SYS_PANIC_MSG_CLEAR, NULL, NULL, NULL) ;
}
```

```
<<sys realm param: constants>>=
#define SYS_PANIC_MSG_CLEAR       10
```

```
<<svc entry: system request functions>>=
[SYS_PANIC_MSG_CLEAR] = sys_realm_panic_msg_clear,
```

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_panic_msg_clear(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_panic_msg_clear(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(req == SYS_PANIC_MSG_CLEAR) ; (void)req ;
    assert(input == NULL) ; (void)input;
    assert(output == NULL) ; (void)output;
    assert(error == NULL) ; (void)error ;

    panic_buf_clear() ;
    return 0 ;
}
```

Print All Panic Messages

```
<<sys svc req: external declarations>>=
extern int
sys_util_panic_msg_print(void) ;
```

The `sys_util_panic_msg_print` function prints to the console all the panic messages which have been accumulated.

Implementation

```
<<sys svc req: external definitions>>=
int
sys_util_panic_msg_print(void)
{
    for (size_t msg_index = 0 ; msg_index < PANIC_BUF_COUNT ; msg_index++) {
        char msg_buf[PANIC_BUF_SIZE] ;
```

```
        int status = sys_panic_msg_get(msg_index, sizeof(msg_buf), msg_buf) ;
        rtcheck_zero_return(status, status) ;

        if (strlen(msg_buf) != 0) {
            puts(msg_buf) ;
        }
    }

    return 0 ;
}
```

Code layout

Panic header file

```
<<panic.h>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   External declarations for panic function
 *--
 */
#ifndef PANIC_H_
#define PANIC_H_

/*
 * Include files
 */
<<panic: interface include files>>
/*
 * Constants
 */
<<panic: constants>>
/*
 * External Functions
 */
<<panic: external declarations>>

#endif /* PANIC_H_ */
```

Panic code file

```
<<panic.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Panic function.
 *--
 */

/*
```

```
 * Include files
 */
#include <stdio.h>
#include <stddef.h>
#include <inttypes.h>
<<panic: include files>>
/*
 * Data Type Declarations
 */
<<panic: data type declarations>>
/*
 * Static Data
 */
<<panic: static data>>
/*
 * External Functions
 */
<<panic: external functions>>
```

Run time checks

It is now time to make explicit the design of internal software error handling. Until now the focus has been on implementing a "short to ground" approach to error situations. Functions supplied to cause a panic condition have a direct path to reset that also leaves context available for diagnostics. Sometimes this approach is called, *fail fast* or *abandonment*. But not every situation encountered is a "panic." It may seem that way now since only system code that insures operations execute according to the needs of the processor core has been considered.

There are three distinct goals for error detection.

Software debugging
: A bug is a bug is a bug and they are always around. The goal is to find them promptly and fix them. Classically in "C", the `assert` macro is used to test expressions that must be true for proper operation.

Runtime violations
: Just as the processor core can cause a divide by zero exception when division instructions are used improperly, some software situations imply similar semantics. It is possible at run time for software to arrive in a state where it no longer knows how to proceed and for which no reasonable recovery can be expected within the context of where the error is discovered. An example of this is in the watchdog timer IRQ handler. The IRQ hander raises a panic if it is unable to queue the background notification. There is little other choice. If the notification is not queued so that background processing can acknowledge the watchdog, the system eventually is reset by the watchdog. In this case, it is better to know that the Background Notification Queue has overflowed than to see a mysterious reset by the watchdog timer.

Recoverable failures
: Some failures have reasonable recovery mechanisms. The archetypical example of opening a file cannot be assumed to always succeed. The file name or other file attributes are simply incorrect input and failure to open a file should be consider in the light of how to handle incorrect external input. Another common situation is dealing with communications. If the protocol governing the communications has recovery mechanisms for lost data, then dropping packet data when there is no storage available can depend upon timeouts and retries of the protocol as a recovery mechanism.

The dividing line between these goals is not sharply defined. Some engineering judgement is required, but setting some policy in this area is essential before amassing significant quantities of code. For our purposes, the judgements have been made in the following manner:

Software debugging
: Classically in "C" programming, the `assert` macro is used state that an expression must be true at some point in the control flow. The use of `static_assert` for compile time checks is similarly motivated for compile time situations.

Whether the assertion check executes is conditionally compiled in or out under the pre-processor symbol, NDEBUG. Traditionally, NDEBUG is undefined during software debugging and testing and release builds define the symbol to remove the check. Although some advocate for leaving assertions in the delivered code, this design assumes that assert should only be used to support debugging and in delivered code the purpose for an assert has been satisfied.

Runtime violations

There are also many circumstances where an explicit test at runtime needs to be made. The "C" language does not automatically support runtime checks. The archetypical example is array bounds checking. No matter, what is missing as a set of runtime checks which remain in the code must be supplied. The usual conventions of having pre-processor function-style macros to save some typing are used.

Recoverable failures

Classically, recoverable failures in "C" code are indicated by returning an encoded integer value to indicate the result status. This can be clumsy if a function needs to return a result in addition to the status. However, it is important to remember that in many cases, a function needs to return both a *result code* and a *result value*. The result code indicates the status of the operation and the result value is the value-result of the operation. There is little mechanism in the language to accomplish this, but there are a couple of means worth mentioning.

- Returning a small structure by value is sometimes a convenient mechanism to return both a result code and a result value. The need to define the structure can be clumsy, but the standard application binary interfaces may make returning small structure values attractive.
- Another common technique, which is sometimes available, is to use part of the value range for a type to indicate errors and the other part to indicate a return value. The classic return of a negative number to indicate errors and non-negative numbers to indicate success is common and used in this design.
- Another interface technique is for the result code to be the return value of the function and the result value is returned by reference using a pointer value passed as an input argument.

Our resolution of these issues here is:

- assert is for debugging and is removed for deployment. An implementation of the assert macro is provided in order to control the ultimate disposition of the assertion.
- A set of runtime check macros is provided, similar in interface to assert, which remain in deployed code. These runtime checks also perform the logical equivalent of an assertion when NDEBUG is not defined, so there is no need to use both. It is only necessary to decide if the expression being asserted is for debugging purposes only, *i.e.* use assert, or is intended for delivered code in which case the runtime check is used and both cases are covered.
- Recoverable failures are handled on a case-by-case basis. Available recovery mechanisms are specific to the application logic or the system services being used.

There is one further complication associated with execution privilege. The same interface to assert and the runtime checks must work for both privileged and unprivileged code. Care must be taken as to how a failing assertion is treated. From unprivileged code, the sys_panic function is invoked as described in the last section. From privileged code, the panic function is invoked. The solution is to provide different entry points for the two execution privileges and use pre-processor macros to make the choice. The pre-processor symbol, PRIVILEGED, is introduced. This symbol is defined when compiling privileged code and the interface selection is based upon whether PRIVILEGED is defined or not. The PRIVILEGED symbol applies to the supplied implementation of assert as well as the runtime checks.

Assert

The assert macro provided by newlib invokes a function named, __assert_func should the assertion fail. A specific implementation of the assert macro and the __assert_func is provided so all the assertion failures are routed ultimately to the panic function.

```
<<assert: unprivileged macros>>=
#ifdef NDEBUG
#   define assert(expr) ((void)0)
```

```
#else
#   define assert(expr)        \
    ((expr) ? (void)0 :         \
    __assert_func (__FILE__, __LINE__, __func__, #expr))
#endif /* NDEBUG */
```

For privileged code, the `__priv_assert_func` is provided as a substitute.

```
<<assert: privileged macros>>=
#ifdef NDEBUG
#   define assert(expr) ((void)0)
#else
#   define assert(expr)         \
    ((expr) ? (void)0 :         \
    __priv_assert_func (__FILE__, __LINE__, __func__, #expr))
#endif /* NDEBUG */
```

```
<<assert.h>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Macros to implement run time condition checks.
 *--
 */
#ifndef ASSERT_H_
#define ASSERT_H_

/*
 * Include files
 */
#include <stdnoreturn.h>
/*
 * Macros
 */
#ifdef PRIVILEGED
extern noreturn void
__priv_assert_func(
    char const *file,
    int line,
    char const *func,
    char const *failedexpr) ;

<<assert: privileged macros>>

#else /* not PRIVILEGED */

extern noreturn void
__assert_func(
    char const *file,
    int line,
    char const *func,
    char const *failedexpr) ;

<<assert: unprivileged macros>>
#endif /* PRIVILEGED */

#if __STDC_VERSION__ >= 201112L
# define static_assert _Static_assert
```

```
#endif

#endif /* ASSERT_H_ */
```

```
<<assert.c>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Replacement function for the newlib "__assert_func" function.
 *--
 */

/*
 * Include files
 */
#include "assert.h"
#include "sys_svc_req.h"
/*
 * Macros
 */
/*
 * External Functions
 */
noreturn void
__assert_func(
    char const *file,
    int line,
    char const *func,
    char const *failedexpr)
{
    static char const assert_msg[] =
            "assertion \"%s\" failed: file \"%s\", line %d, function: %s\n" ;

    sys_panic(assert_msg, failedexpr, file, line, func) ;
}
```

```
<<priv_assert.c>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Replacement function for the newlib "__assert_func" function.
 *--
 */

/*
 * Include files
 */
#include "assert.h"
#include "panic.h"
/*
 * Macros
 */
/*
```

```
 * External Functions
 */
noreturn void
__priv_assert_func(
    char const *file,
    int line,
    char const *func,
    char const *failedexpr)
{
    static char const assert_msg[] =
            "assertion \"%s\" failed: file \"%s\", line %d, function: %s\n" ;

    panic(assert_msg, failedexpr, file, line, func) ;
}
```

Runtime checks

Runtime checks are intended to remain in the deployed code. In newer or more controlling languages, many of these checks are provided by the language. A number of different checks tailored to specific usage are provided. For example, range checking for array indices is common enough to warrant its own macro.

Following the pattern for `assert`, both privileged and unprivileged versions are defined. The only difference is how the failure situation is handled.

Panic runtime checks

The runtime checks in this group all raise a **panic condition**.

```
<<rtcheck: unprivileged macros>>=
#define rtcheck(expr)       \
    (expr) ? (void)0 : rtcheck_panic(#expr, __FILE__, __LINE__, __func__)
```

expr
: A “C” expression which is expected to evaluate as true.

The `rtcheck` macro tests if `expr` is true. If `expr` is true, then execution continues. Otherwise, a **panic** condition is raised.

```
<<rtcheck: unprivileged macros>>=
#define rtcheck_max(expr, maxvalue)         \
    ((expr) < (maxvalue)) ? (void)0 :       \
        rtcheck_panic(#expr " < " #maxvalue, __FILE__, __LINE__, __func__)
```

expr
: A “C” expression which is compared.

maxvalue
: A “C” expression which defines the maximum value expected.

The `rtcheck_max` macro evaluates `expr` to determine if it is **less than** `maxvalue`. If so execution continues. Otherwise, a **panic** condition is raised. *N.B.* the comparison is strictly less than.

```
<<rtcheck: unprivileged macros>>=
#define rtcheck_min(expr, minvalue)          \
    ((expr) >= (minvalue)) ? (void)0 :       \
        rtcheck_panic(#expr " >= " #minvalue, __FILE__, __LINE__, __func__)
```

expr
A "C" expression which is compared.

minvalue
A "C" expression which defines the minimum value expected.

The `rtcheck_min` macro evaluates `expr` to determine if it is **greater than or equal to** `minvalue`. If so execution continues. Otherwise, a **panic** condition is raised. *N.B.* the comparison is greater than or equal to.

```
<<rtcheck: unprivileged macros>>=
#define rtcheck_range(expr, minvalue, maxvalue)                              \
    ((expr) >= (minvalue) && (expr) < (maxvalue)) ? (void)0 :                \
        rtcheck_panic(#expr " >= " #minvalue " && " #expr " < " #maxvalue,   \
                __FILE__, __LINE__, __func__)
```

expr
A "C" expression which is compared.

minvalue
A "C" expression which defines the minimum value expected.

maxvalue
A "C" expression which defines the maximum value expected.

The `rtcheck_range` macro combines the tests of `rtcheck_min` and `rtcheck_max`. It evaluates `expr` to be **greater than or equal to** `minvalue` and **less than** `maxvalue`.

N.B. this macro evaluates `expr` multiple times and so it is important that `expr` contain no side effects.

The following definitions contain the privileged version of the previous runtime checks.

```
<<rtcheck: privileged macros>>=
#define rtcheck(expr)        \
    (expr) ? (void)0 : priv_rtcheck_panic(#expr, __FILE__, __LINE__, __func__)

#define rtcheck_max(expr, maxvalue)          \
    ((expr) < (maxvalue)) ? (void)0 :        \
        priv_rtcheck_panic(#expr " < " #maxvalue, __FILE__, __LINE__, __func__)

#define rtcheck_min(expr, minvalue)          \
    ((expr) >= (minvalue)) ? (void)0 :       \
        priv_rtcheck_panic(#expr " >= " #minvalue, __FILE__, __LINE__, __func__)

#define rtcheck_range(expr, minvalue, maxvalue)                              \
    ((expr) >= (minvalue) && (expr) < (maxvalue)) ? (void)0 :                \
        priv_rtcheck_panic(#expr " >= " #minvalue " && " #expr " < " #maxvalue,   \
                __FILE__, __LINE__, __func__)
```

Returning runtime checks

On many occasions, a runtime check should *not* raise a panic condition. In this section, a series of macros similar to those above are given. These macros execute a `return` statement rather than causing a panic. Some care must be used with these macros. They must only be used where a `return` statement is acceptable.

A few of points to note about the implementation:

- The mnemonic way to read the macro is, "run time check the condition for being true and otherwise return the given value."
- The macros expand to a `do ... while (0)` statement in order to insure that the expansion creates only a single "C" statement.
- Each macro also includes an invocation of `assert` to stop execution before the return when debugging. The situations where these macros are used are considered an error.
- There are added macros for common comparisons such as zero and `NULL`.
- The `expr` macro arguments are evaluated multiple times in the macros. So, `expr` must not contain side effects.

```
<<rtcheck: return macros>>=
#define rtcheck_return(expr, retvalue)      \
    do {                                    \
        assert(expr) ;                      \
        if (!(expr)) {                      \
            return retvalue ;               \
        }                                   \
    } while (0)
```

expr
: A "C" expression which is expected to evaluate as true.

retvalue
: A "C" expression the value of which is returned if `expr` is false.

The `rtcheck_return` macro evaluates `expr`. If `expr` evaluates to `true` then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

```
<<rtcheck: return macros>>=
#define rtcheck_max_return(expr, maxvalue, retvalue)        \
    do {                                                    \
        assert ((expr) < (maxvalue)) ;                      \
        if (!((expr) < (maxvalue))) {                       \
            return retvalue ;                               \
        }                                                   \
    } while (0)
```

expr
: A "C" expression which is compared.

maxvalue
: A "C" expression which defines the maximum value expected.

retvalue
: A "C" expression the value of which is returned if the comparison fails.

The `rtcheck_max_return` macro evaluates `expr` to determine if it **less than** `maxvalue`. If so, then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

```
<<rtcheck: return macros>>=
#define rtcheck_min_return(expr, minvalue, retvalue)        \
    do {                                                    \
        assert ((expr) >= (minvalue)) ;                     \
        if (!((expr) >= (minvalue))) {                      \
            return retvalue ;                               \
        }                                                   \
    } while (0)
```

expr
A "C" expression which is compared.

minvalue
A "C" expression which defines the minimum value expected.

retvalue
A "C" expression the value of which is returned if the comparison fails.

The `rtcheck_min_return` macro evaluates `expr` to determine if it **greater than or equal to** `minvalue`. If so, then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

```
<<rtcheck: return macros>>=
#define rtcheck_range_return(expr, minvalue, maxvalue, retvalue)    \
    do {                                                            \
        assert ((expr) >= (minvalue) && (expr) < (maxvalue)) ;      \
        if (!((expr) >= (minvalue) && (expr) < (maxvalue))) {       \
            return retvalue ;                                       \
        }                                                           \
    } while (0)
```

expr
A "C" expression which is compared.

minvalue
A "C" expression which defines the minimum value expected.

maxvalue
A "C" expression which defines the maximum value expected.

retvalue
A "C" expression the value of which is returned if the comparison fails.

The `rtcheck_range_return` macro combines the tests of `rtcheck_min_return` and `rtcheck_max_return`. It evaluates `expr` to determine if it is **greater than or equal to** `minvalue` and **less than** `maxvalue`. If so, then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

N.B. this macro evaluates `expr` multiple times and so it is important that `expr` contain no side effects.

```
<<rtcheck: return macros>>=
#define rtcheck_zero_return(expr, retvalue)     \
    do {                                        \
        assert ((expr) == 0) ;                  \
        if (!((expr) == 0)) {                   \
            return retvalue ;                   \
        }                                       \
    } while (0)
```

expr
A “C” expression which is compared to zero.

retvalue
A “C” expression the value of which is returned if the comparison fails.

The `rtcheck_zero_return` macro compares `expr` to zero. If `expr` is equal to zero, then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

```
<<rtcheck: return macros>>=
#define rtcheck_not_zero_return(expr, retvalue) \
    do {                                        \
        assert ((expr) != 0) ;                  \
        if (!((expr) != 0)) {                   \
            return retvalue ;                   \
        }                                       \
    } while (0)
```

expr
A “C” expression which is compared to zero.

retvalue
A “C” expression the value of which is returned if the comparison fails.

The `rtcheck_not_zero_return` macro compares `expr` to zero. If `expr` is not equal to zero, then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

```
<<rtcheck: return macros>>=
#define rtcheck_not_negative_return(expr, retvalue) \
    do {                                        \
        assert ((expr) >= 0) ;                  \
        if (!((expr) >= 0)) {                   \
            return retvalue ;                   \
        }                                       \
    } while (0)
```

expr
A “C” expression which is compared to zero.

retvalue
A “C” expression the value of which is returned if the comparison fails.

The `rtcheck_not_negative_return` macro compares `expr` to zero. If `expr` is **greater than or equal to** zero, then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

```
<<rtcheck: return macros>>=
#define rtcheck_NULL_return(expr, retvalue)     \
    do {                                        \
        assert ((expr) == NULL) ;               \
        if (!((expr) == NULL)) {                \
            return retvalue ;                   \
        }                                       \
    } while (0)
```

expr
A “C” expression which evaluates to a pointer type.

retvalue
A “C” expression the value of which is returned if the comparison fails.

The `rtcheck_NULL_return` macro compares `expr` to the `NULL` pointer value. If `expr` is **equal to** `NULL`, then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

```
<<rtcheck: return macros>>=
#define rtcheck_not_NULL_return(expr, retvalue) \
    do {                                        \
        assert ((expr) != NULL) ;               \
        if (!((expr) != NULL)) {                \
            return retvalue ;                   \
        }                                       \
    } while (0)
```

expr
A “C” expression which evaluates to a pointer type.

retvalue
A “C” expression the value of which is returned if the comparison fails.

The `rtcheck_not_NULL_return` macro compares `expr` to the `NULL` pointer value. If `expr` is **not equal to** `NULL`, then execution continues. Otherwise, a `return` statement is executed with `retvalue` as its operand.

Code Layout

```
<<rtcheck.h>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Macros to implement run time condition checks.
 *--
 */
#ifndef RTCHECK_H_
#define RTCHECK_H_

/*
 * Include files
 */
#include <stdnoreturn.h>
```

```
#include <assert.h>
/*
 * Macros
 */
#ifdef PRIVILEGED
extern noreturn void
priv_rtcheck_panic(
    char const *failedexpr,
    char const *file,
    int line,
    char const *func) ;
<<rtcheck: privileged macros>>

#else /* not PRIVILEGED */

extern noreturn void
rtcheck_panic(
    char const *failedexpr,
    char const *file,
    int line,
    char const *func) ;

<<rtcheck: unprivileged macros>>
#endif /* PRIVILEGED */

<<rtcheck: return macros>>
/*
 * External Functions
 */

#endif /* RTCHECK_H_ */
```

```
<<rtcheck.c>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Function to handle run time checks for unprivileged code.
 *--
 */
/*
 * Include files
 */
#include "rtcheck.h"
#include "sys_svc_req.h"
/*
 * Macros
 */
/*
 * External Functions
 */
noreturn void
rtcheck_panic(
    char const *failedexpr,
    char const *file,
    int line,
    char const *func)
{
    static char const rtcheck_msg[] =
```

```
            "run time check \"%s\" failed: file \"%s\", line %d, function: %s\n" ;

    sys_panic(rtcheck_msg, failedexpr, file, line, func) ;
}
```

```
<<priv_rtcheck.c>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Function to handle run time checks for privileged code.
 *--
 */
/*
 * Include files
 */
#include "rtcheck.h"
#include "panic.h"
/*
 * Macros
 */
/*
 * External Functions
 */
noreturn void
priv_rtcheck_panic(
    char const *failedexpr,
    char const *file,
    int line,
    char const *func)
{
    static char const rtcheck_msg[] =
            "run time check \"%s\" failed: file \"%s\", line %d, function: %s\n" ;

    panic(rtcheck_msg, failedexpr, file, line, func) ;
}
```

Missing exception handler

The default exception handler defined previously provided a better way to handle exceptions than the default handlers usually provided in CMSIS start up files. That exception handler deferred the work of handling a missing exception to a function named, SystemMissingHandler(). The symbol for the function is defined as *weak* so it can be overridden. The default implementation of the function just invokes `SystemAbend()`. Abnormal endings break into the debugger if it is attached and reset the system otherwise. This behavior does cover all the *necessary* conditions to prevent the processor core from running away and potentially locking up.

Neither of the actions of `SystemAbend()` is *sufficient* for a robust system. If you happen to have an attached debugger, execution stops and your debugger may display some useful information, but it is still necessary to examine a number of core system registers to determine the exact cause of the exception. Code is needed to perform that examination and report the information in a convenient form. If there is no debugger attached, then the system is reset and the reason for the reset remains a mystery. Capturing the circumstances that have caused a reset is an essential component for a post-mortem examination.

In this section, a replacement for the default `SystemMissingHandler()` function is developed that solves the need for detailed information about system faults and missing IRQ handlers. To be clear, when an exception is triggered for which no specific handler has been put into vector table, then the following is required:

- Capture the essential information about the exception in a manner that supports post-mortem examination.
- Format and print the captured information if a debugger is present.

Capturing the exception information breaks down into two parts:

1. Reading the system registers which contain fault information and
2. Storing the register values somewhere which is accessible after a reset.

The missing exception handler presented also handles the major system faults, *i.e.* HardFault, BusFault, MemManageFault, and UsageFault. These are important system faults as they indicate that the processor has detected misbehavior on the part of the program or the operating environment. These faults are considered unrecoverable, *i.e.* they must ultimately end in an abnormal termination. That approach makes the system faults good candidates to handle as a missing exception. Some systems may need to attempt some form of recovery. In those cases, the default fault handler for the fault can be overridden to supply any required recovery processing.

Faulting Information

The following data structure defines the information collected when a missing exception occurs. Note that it is not specific to the type of fault. Rather, all the information which might pertain to diagnosing the cause of the fault is gathered and post-mortem processing must determine what is applicable.

Fault status structure declaration

```
<<missing exceptions: data type declarations>>=
typedef struct {
    uint32_t alloc_count ;                  // ❶
    uint64_t timestamp ;
    ExceptionFrame exc_frame ;
    uint32_t exc_return ;
    uint32_t ipsr ;
    uint32_t cfsr ;
    uint32_t hfsr ;
    uint32_t dfsr ;
    uint32_t mmfar ;
    uint32_t bfar ;
    uint32_t apollo3_fsr ;                  // ❷
    uint32_t icode_far ;
    uint32_t dcode_far ;
    uint32_t sys_far ;
    uint32_t reset_status ;
} FaultStatus ;
```

❶ This is a simple incrementing counter which counts the number of missed exceptions since the last power on reset.

❷ The Apollo 3 has some additional fault address registers that record fault circumstances specific to the SOC.

The members of the previous structure are a mix of processor registers and Apollo 3 SOC specific registers. The structure members are named for the corresponding register that they hold.

```
<<missing exceptions: include files>>=
#include "system_apollo3.h"
```

Fault Status Storage

In Chapter 2, linker sections that survive reset were set up and they are only initialized at power up. That memory is used to store the exception information.

First, a array of `FaultStatus` elements to serve as a storage pool is allocated.

```
<<missing exceptions: static data>>=
__attribute__((section(".noinit_bss")))
static FaultStatus fault_status_pool[FAULT_STATUS_POOL_SIZE] ;
```

A data structure to track allocation of elements from the storage pool is required. The storage pool is treated as a circular queue where overflow is allowed. This results in the last `FAULT_STATUS_POOL_SIZE` number of fault statuses being stored.

One particular, albeit rare, circumstance must be accommodated. It is possible for a system to get into a continual reset situation. Consider the following scenario. A system fault causes the execution to store away the status information and reset. The fault might be caused by a hardware malfunction. If the failure mode persists, the system may run for a short while and encounter the same situation which caused the original fault. This would cause another set of fault status information to be recorded and the processor would be reset. If this happens repeatedly, you end up with an overflowing fault status storage pool with very low quality information. It is likely that all the entries are the same except for the timestamp. To avoid this rare circumstance, the allocation code insures that that the fault status overflow does not consume the status slot that was first in the sequence. Fault status slots are allocated in a circular, overflowing manner, but the first status slot is prevented from being reused. This means the first fault is recorded and if the reset continues to repeat, the later instances are not allowed to overwrite the first recorded fault status. Having access to the first instance can indicate the triggering situation for a post-mortem evaluation of repeated resets.

The implementation of the design concept to preserve the first fault status is accomplished by using a pointer into the storage where the circular wrap happens. So, rather than wrapping back to the beginning of the storage pool when the pool storage bounds are exceeded, the wrap back is to the second slot. This can be generalized to keep the first N occurrence by specifying the location of the wrap in the pool. In this implementation, only the first occurrence is saved.

Some systems may need to catch repeated reset scenarios and take action to completely shut down the system. This might be required to prevent damage to the either the system itself or damage to the environment where the system interacts. In those cases, a *rate* of reset may be calculated (from the timestamp and count information) and if it exceeds a threshold, additional actions can be taken to break the cycle and place the system in a mode where the resets stop. One possibility is to disable all interrupts and place the system in its lowest power mode. A continual reset cycle left unchecked can easily deplete a battery powered system. Such requirements are more stringent than implemented here, but the `alloc_count` and `timestamp` members hold the data required for reasoning about the rate of resets encountered.

The data structure used to implement an overflowing circular queue which leaves the first element in place follows. The additional element required is a place in the queue where the wrap around happens.

```
<<missing exceptions: static data>>=
__attribute__((section(".noinit_data")))
static struct {
    FaultStatus *const pool_start ;
    FaultStatus *const pool_wrap ;                          // ❶
    FaultStatus *next ;
    uint32_t alloc_count ;                                  // ❷
} fault_status_allocator = {
    .pool_start = fault_status_pool,
    .pool_wrap = fault_status_pool + 1,
    .next = fault_status_pool,
    .alloc_count = 0,
} ;
```

❶ The value for the `next` pointer when it wraps around.

❷ An sequential counter to keep track of the total number of fault status allocations.

A default size for the status storage pool is provided with the usual flexibility to define a different pool size by changing a pre-processor symbol.

```
<<missing exceptions: constants>>=
#ifndef FAULT_STATUS_POOL_SIZE
#   define  FAULT_STATUS_POOL_SIZE    4
#endif /* FAULT_STATUS_POOL_SIZE */

static_assert(FAULT_STATUS_POOL_SIZE > 1,
        "fault status storage pool size must be at least 2") ;
```

Fault Status Allocation

```
<<missing exceptions: forward references>>=
static FaultStatus *
fsb_alloc(void) ;
```

The `fsb_alloc` function allocates a fault status block from the fault status pool.

The implementation follows from the previous discussion and is another example of using type specific memory allocation from fixed storage pools rather than from a system heap.

Implementation

```
<<missing exceptions: static functions>>=
static FaultStatus *
fsb_alloc(void)
{
    FaultStatus *slot = fault_status_allocator.next ;

    fault_status_allocator.next += 1 ;
    if (fault_status_allocator.next >= fault_status_pool + FAULT_STATUS_POOL_SIZE) {
        fault_status_allocator.next = fault_status_allocator.pool_wrap ;            // ❶
    }

    fault_status_allocator.alloc_count += 1 ;
    slot->alloc_count = fault_status_allocator.alloc_count ;

    return slot ;
}
```

❶ Wrapping to a location other than the start of the storage pool gives the ability to leave initial fault information untouched on overflow.

`SystemMissingHandler()`

The implementation of the missing exception handler is three steps.

1. Capture the fault status information.
2. Print the information if a debugger is present.
3. Terminate the program abnormally.

```
<<missing exceptions: external functions>>=
noreturn void
SystemMissingHandler(
    ExceptionFrame *const exc_frame,
    uint32_t exc_return,
    uint32_t ipsr)
{
    FaultStatus *status = capture_fault_status(exc_frame, exc_return, ipsr) ;

    if (stwd_debug_enabled()) {
        print_fault_status(status) ;
    }

    SystemAbend() ;
}
```

exc_frame
A pointer to an exception frame. Typically this argument points into the stack area populated when the exception was raised.

exc_return
The exception return value placed in the Link Register when the exception was raised.

ipsr
The value of the IPSR register showing the exception number.

The function prototypes for the system register bit twiddling code are needed.

```
<<missing exceptions: include files>>=
#include "sys_twiddle.h"
```

Allocate a `FaultStatus` object and initialize it from the values of system registers.

```
<<missing exceptions: forward references>>=
static FaultStatus *
capture_fault_status(
    ExceptionFrame *const exc_frame,
    uint32_t exc_return,
    uint32_t ipsr) ;
```

exc_frame
A pointer to an exception frame. Typically this argument points into the stack area populated when the exception was raised.

exc_return
The exception return value placed in the Link Register when the exception was raised.

ipsr
The value of the IPSR register showing the exception number.

Handling the FPU context is one complication in the implementation of `capture_fault_status`. The FPU context determines if the exception frame contains the values of FPU registers. In this particular case, it will not, since the disabled FPU register stacking for exceptions. has been disabled. However, it is reasonable to implement the check in case the FPU register stacking policy changes.

```
<<missing exceptions: static functions>>=
static FaultStatus *
capture_fault_status(
```

```
    ExceptionFrame *const exc_frame,
    uint32_t exc_return,
    uint32_t ipsr)
{
    FaultStatus *status = fsb_alloc() ;

    status->timestamp = dtwd_get_timestamp() ;
    if (stwd_is_fpu_context_active()) {
        status->exc_frame = *exc_frame ;
    } else {
        memset(&status->exc_frame, 0, sizeof(status->exc_frame)) ;
        memcpy(&status->exc_frame, exc_frame,
                offsetof(ExceptionFrame, xpsr) + sizeof(uint32_t)) ;     // ❶
    }
    status->exc_return = exc_return ;
    status->ipsr = ipsr ;
    status->cfsr = SCB->CFSR ;
    status->hfsr = SCB->HFSR ;
    status->dfsr = SCB->DFSR ;
    status->mmfar = SCB->MMFAR ;
    status->bfar = SCB->BFAR ;
    status->apollo3_fsr = MCUCTRL->FAULTSTATUS ;
    status->icode_far = MCUCTRL->ICODEFAULTADDR ;
    status->dcode_far = MCUCTRL->DCODEFAULTADDR ;
    status->sys_far = MCUCTRL->SYSFAULTADDR ;
    MCUCTRL->FAULTSTATUS = btwd_mask(3, 0) ;                             // ❷
    status->reset_status = noinit_status.reset_status ;                  // ❸

    return status ;
}
```

❶ There are eight registers pushed in a basic exception frame. Note advantage is taken of having defined an `ExceptionFramed` as a “packed” structure.

❷ The data sheet specifies that any write to this register clears *all* the status bits. All the status bits are written just to make sure.

❸ The reset status from the last time the processor reset is recorded. Recall that this status is recorded into a variable at reset time and is overwritten at each reset. This saves the value of the last reset status before purposely reseting the processor.

Since the reset status from the last reset is accessed, a header file declares the `noinit_status` variable.

```
<<missing exceptions: include files>>=
#include "linker_symbols.h"
```

A timestamp our fault status is also used.

```
<<missing exceptions: include files>>=
#include "dev_twiddle.h"
```

Format and print the fault status to the standard output.

```
<<missing exceptions: forward references>>=
static void
print_fault_status(
    FaultStatus *const status) ;
```

status
A pointer to a fault status object containing the faulting information to be printed.

The implementation of `print_fault_status` breaks down to printing the common information followed by information specific to the type of the fault.

```
<<missing exceptions: static functions>>=
static void
print_fault_status(
    FaultStatus *const status)
{
    print_exception_frame(status) ;
    switch (status->ipsr) {
    case HARDFAULT_EXC_NUMBER:
        print_hard_fault_status(status) ;
        break ;

    case MEMMANAGE_EXC_NUMBER:
        print_memmanage_fault_status(status) ;
        break ;

    case BUSFAULT_EXC_NUMBER:
        print_bus_fault_status(status) ;
        break ;

    case USAGEFAULT_EXC_NUMBER:
        print_usage_fault_status(status) ;
        break ;

    case DEBUGMONITOR_EXC_NUMBER:
        print_debug_fault_status(status) ;
        break ;
    /*
     * N.B. no default case. Only the five system generated exceptions have
     * any special treatment.
     */
    }
}
```

The exception numbers in the IPSR register are different from the *irq numbers* of the exceptions. The exception number value is the IRQ number value plus 16 (recall that IRQ numbers for system exceptions are negative)[1]. The following are the exception numbers of interest here.

```
<<missing exceptions: constants>>=
#define HARDFAULT_EXC_NUMBER    ((int)HardFault_IRQn + 16)
#define MEMMANAGE_EXC_NUMBER    ((int)MemoryManagement_IRQn + 16)
#define BUSFAULT_EXC_NUMBER     ((int)BusFault_IRQn + 16)
#define USAGEFAULT_EXC_NUMBER   ((int)UsageFault_IRQn + 16)
#define DEBUGMONITOR_EXC_NUMBER   ((int)DebugMonitor_IRQn + 16)
```

The IRQ numbers are defined in the device CMSIS header file.

```
<<missing exceptions: include files>>=
#include "apollo3.h"
```

[1] Yes, it is all a bit confusing. Just try to keep in mind that exceptions are the general terms and IRQ's are specific to peripheral devices and they all have to share the exception vector table.

Format and print common exception frame information to the standard output.

```
<<missing exceptions: forward references>>=
static void
print_exception_frame(
    FaultStatus *const status) ;
```

status
A pointer to a fault status object containing the faulting information to be printed.

The implementation is uncomplicated, but does pass off to other functions printing specific types of registers.

```
<<missing exceptions: static functions>>=
static void
print_exception_frame(
    FaultStatus *const status)
{
    print_exception_name(status->ipsr) ;
    printf("COUNT: %" PRIu32 "\n", status->alloc_count) ;

    uint32_t time_int = status->timestamp >> 15 ;
    uint32_t time_frac = status->timestamp & btwd_mask(15, 0) ;
    uint32_t time_ms = (time_frac * UINT32_C(1000)) >> 15 ;
    printf("TIME: %" PRIu32 ".%03" PRIu32 "\n", time_int, time_ms) ;

    printf("R0: 0x%08" PRIx32 "\n", status->exc_frame.r0) ;
    printf("R0: 0x%08" PRIx32 "\n", status->exc_frame.r0) ;
    printf("R1: 0x%08" PRIx32 "\n", status->exc_frame.r1) ;
    printf("R2: 0x%08" PRIx32 "\n", status->exc_frame.r2) ;
    printf("R3: 0x%08" PRIx32 "\n", status->exc_frame.r3) ;
    printf("IP: 0x%08" PRIx32 "\n", status->exc_frame.ip) ;
    printf("LR: 0x%08" PRIx32 "\n", status->exc_frame.lr) ;
    printf("PC: 0x%08" PRIx32 "\n", status->exc_frame.pc) ;
    print_xpsr(status->exc_frame.xpsr) ;
    print_exc_return(status->exc_return) ;
    print_reset_status(status->reset_status) ;
}
```

Print the name of the exception identified by an exception number.

```
<<missing exceptions: forward references>>=
static void
print_exception_name(
    uint32_t ipsr) ;
```

ipsr
The value of the IPSR portion of the processor status register.

The implementation of the `print_exception_frame` follows a conventional mapping of numbers to strings using an array. The numeric IRQ values are sequential and suitable to use as an array index. The IRQ numeric values are given in comments to help with the array layout.

```
<<missing exceptions: static functions>>=
static void
print_exception_name(
    uint32_t ipsr)
{
    static char const reserved_name[] = "Reserved" ;
```

```
    static char const *const irq_names[] = {
        reserved_name,          //  0
        "Reset",                //  1
        "NMI",                  //  2
        "HardFault",            //  3
        "MemManage",            //  4
        "BusFault",             //  5
        "UsageFault",           //  6
        reserved_name,          //  7
        reserved_name,          //  8
        reserved_name,          //  9
        reserved_name,          // 10
        "SVCall",               // 11
        "Reserved Debug",       // 12
        reserved_name,          // 13
        "PendSV",               // 14
        "SysTick",              // 15
        "BROWNOUT",             // 16
        "WDT",                  // 17
        "RTC",                  // 18
        "VCOMP",                // 19
        "IOSLAVE",              // 20
        "IOSLAVEACC",           // 21
        "IOMSTR0",              // 22
        "IOMSTR1",              // 23
        "IOMSTR2",              // 24
        "IOMSTR3",              // 25
        "IOMSTR4",              // 26
        "IOMSTR5",              // 27
        "BLE",                  // 28
        "GPIO",                 // 29
        "CTIMER",               // 30
        "UART0",                // 31
        "UART1",                // 32
        "SCARD",                // 33
        "ADC",                  // 34
        "PDM",                  // 35
        "MSPI0",                // 36
        reserved_name,          // 37
        "STIMER",               // 38
        "STIMER_CMPR0",         // 39
        "STIMER_CMPR1",         // 40
        "STIMER_CMPR2",         // 41
        "STIMER_CMPR3",         // 42
        "STIMER_CMPR4",         // 43
        "STIMER_CMPR5",         // 44
        "STIMER_CMPR6",         // 45
        "STIMER_CMPR7",         // 46
        "CLKGEN",               // 47
    } ;

    if (ipsr >= COUNTOF(irq_names)) {
        printf("\nUncaught Exception: %" PRIu32 "\n", ipsr) ;
    } else {
        printf("\nUncaught Exception: %s (%" PRIu32 ")\n", irq_names[ipsr], ipsr) ;
    }
}
```

```
<<missing exceptions: forward references>>=
static void
print_xpsr(
```

```
    uint32_t xpsr) ;
```

```
<<missing exceptions: static functions>>=
static void
print_xpsr(
    uint32_t xpsr)
{
    printf("xPSR: 0x%08" PRIx32 "\n    ", xpsr) ;
    printf("N: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(xpsr, xPSR_N)) ;
    printf("Z: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(xpsr, xPSR_Z)) ;
    printf("C: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(xpsr, xPSR_C)) ;
    printf("V: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(xpsr, xPSR_V)) ;
    printf("Q: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(xpsr, xPSR_Q)) ;
    printf("ISR: %" PRIu32 "\n", BTWD_FIELD_EXTRACT(xpsr, xPSR_ISR)) ;
}
```

```
<<missing exceptions: forward references>>=
static void
print_exc_return(
    uint32_t exc_return) ;
```

```
<<missing exceptions: static functions>>=
static void
print_exc_return(
    uint32_t exc_return)
{
    printf("EXC_RETURN: 0x%08" PRIx32 "\n    ", exc_return) ;
    printf("Stack: %s, ", ((exc_return & 0x04) == 0) ? "MSP" : "PSP") ;
    printf("Mode: %s, ", ((exc_return & 0x08) == 0) ? "Handler" : "Thread") ;
    printf("Frame: %s\n", ((exc_return & 0x10) == 0) ? "FPU" : "Basic") ;
}
```

Format and print the bit fields of the Apollo 3 Reset Generator peripheral `STAT` register.

```
<<missing exceptions: forward references>>=
static void
print_reset_status(
    uint32_t reset_status) ;
```

reset_status
The value of the reset generator `STAT` register for an Apollo 3 SOC.

```
<<missing exceptions: static functions>>=
static void
print_reset_status(
    uint32_t reset_status)
{
    static char const *const status_names[] = {
        "EXR", "POR", "BOR", "SWR", "POIR", "DBGR", "WDR",
        "BOU", "BOC", "BOF", "BOB"
    } ;
    printf("Reset Status: 0x%08" PRIx32 "\n    ", reset_status) ;

    uint32_t status_mask = 1 ;
    bool first = true ;
    char const *const *const names_end = status_names + COUNTOF(status_names) ;
    for (char const *const *names_iter = status_names ;
            names_iter < names_end ; names_iter++) {
        if ((reset_status & status_mask) != 0) {
```

```
            if (first) {
                printf("%s", *names_iter) ;
                first = false ;
            } else {
                printf(", %s", *names_iter) ;
            }
        }
        status_mask <<= 1 ;
    }
    printf("\n") ;
}
```

```
<<missing exceptions: forward references>>=
static void
print_hard_fault_status(
    FaultStatus *const status) ;
```

```
<<missing exceptions: static functions>>=
static void
print_hard_fault_status(
    FaultStatus *const status)
{
    uint32_t hfsr = status->hfsr ;
    printf("HFSR: 0x%08" PRIx32 "\n    ", hfsr) ;
    printf("DEBUGEVT: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(hfsr, SCB_HFSR_DEBUGEVT)) ;
    printf("FORCED: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(hfsr, SCB_HFSR_FORCED)) ;
    printf("VECTTBL: %" PRIu32 "\n", BTWD_FIELD_EXTRACT(hfsr, SCB_HFSR_VECTTBL)) ;
    print_memmanage_fault_status(status) ;
    print_bus_fault_status(status) ;
    print_usage_fault_status(status) ;
    print_debug_fault_status(status) ;
    print_apollo3_fault_status(status) ;
}
```

```
<<missing exceptions: forward references>>=
static void
print_debug_fault_status(
    FaultStatus *const status) ;
```

```
<<missing exceptions: static functions>>=
static void
print_debug_fault_status(
    FaultStatus *const status)
{
    uint32_t dfsr = status->dfsr ;
    printf("DFSR: 0x%08" PRIx32 "\n    ", dfsr) ;
    printf("EXTERNAL: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(dfsr, SCB_DFSR_EXTERNAL)) ;
    printf("VCATCH: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(dfsr, SCB_DFSR_VCATCH)) ;
    printf("DWTTRAP: %" PRIu32 "\n", BTWD_FIELD_EXTRACT(dfsr, SCB_DFSR_DWTTRAP)) ;
    printf("BKPT: %" PRIu32 "\n", BTWD_FIELD_EXTRACT(dfsr, SCB_DFSR_BKPT)) ;
    printf("HALTED: %" PRIu32 "\n", BTWD_FIELD_EXTRACT(dfsr, SCB_DFSR_HALTED)) ;
}
```

```
<<missing exceptions: forward references>>=
static void
print_memmanage_fault_status(
    FaultStatus *const status) ;
```

```
<<missing exceptions: static functions>>=
```

```
static void
print_memmanage_fault_status(
    FaultStatus *const status)
{
    uint32_t cfsr = status->cfsr ;
    uint32_t mmfsr = BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_MEMFAULTSR) ;
    printf("MMFSR: 0x%08" PRIx32 "\n    ", mmfsr) ;
    printf("MMARVALID: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_MMARVALID)) ;
    printf("MSTKERR: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_MSTKERR)) ;
    printf("MUNSTKERR: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_MUNSTKERR)) ;
    printf("DACCVIOL: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_DACCVIOL)) ;
    printf("IACCVIOL: %" PRIu32 "\n", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_IACCVIOL)) ;

    printf("MMFAR: 0x%08" PRIx32 "\n", status->mmfar) ;
    print_apollo3_fault_status(status) ;
}
```

```
<<missing exceptions: forward references>>=
static void
print_bus_fault_status(
    FaultStatus *const status) ;
```

```
<<missing exceptions: static functions>>=
static void
print_bus_fault_status(
    FaultStatus *const status)
{
    uint32_t cfsr = status->cfsr ;
    uint32_t bfsr = BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_BUSFAULTSR) ;
    printf("BFSR: 0x%08" PRIx32 "\n    ", bfsr) ;
    printf("BFARVALID: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_BFARVALID)) ;
    printf("LSPERR: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_LSPERR)) ;
    printf("STKERR: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_STKERR)) ;
    printf("UNSTKERR: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_UNSTKERR)) ;
    printf("IMPRECISERR: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_IMPRECISERR)) ;
    printf("PRECISERR: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_PRECISERR)) ;
    printf("IBUSERR: %" PRIu32 "\n", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_IBUSERR)) ;

    printf("BFAR: 0x%08" PRIx32 "\n", status->bfar) ;
    print_apollo3_fault_status(status) ;
}
```

```
<<missing exceptions: forward references>>=
static void
print_apollo3_fault_status(
    FaultStatus *const status) ;
```

```
<<missing exceptions: static functions>>=
static void
print_apollo3_fault_status(
    FaultStatus *const status)
{
    uint32_t apollo3_fsr = status->apollo3_fsr ;

    if (BTWD_FIELD_TEST(apollo3_fsr, MCUCTRL_FAULTSTATUS_ICODEFAULT)) {
        printf("ICODEFAULT: 0x%08" PRIx32 "\n", status->icode_far) ;
    }
    if (BTWD_FIELD_TEST(apollo3_fsr, MCUCTRL_FAULTSTATUS_DCODEFAULT)) {
        printf("DCODEFAULT: 0x%08" PRIx32 "\n", status->dcode_far) ;
    }
```

```
    if (BTWD_FIELD_TEST(apollo3_fsr, MCUCTRL_FAULTSTATUS_SYSFAULT)) {
        printf("SYSFAULT: 0x%08" PRIx32 "\n", status->sys_far) ;
    }
}
```

```
<<missing exceptions: forward references>>=
static void
print_usage_fault_status(
    FaultStatus *const status) ;
```

```
<<missing exceptions: static functions>>=
static void
print_usage_fault_status(
    FaultStatus *const status)
{
    uint32_t cfsr = status->cfsr ;
    uint32_t ufsr = BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_USGFAULTSR) ;
    printf("UFSR: 0x%08" PRIx32 "\n     ", ufsr) ;
    printf("DIVBYZERO: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_DIVBYZERO)) ;
    printf("UNALIGNED: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_UNALIGNED)) ;
    printf("NOCP: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_NOCP)) ;
    printf("INVPC: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_INVPC)) ;
    printf("INVSTATE: %" PRIu32 ", ", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_INVSTATE)) ;
    printf("UNDEFINSTR: %" PRIu32 "\n", BTWD_FIELD_EXTRACT(cfsr, SCB_CFSR_UNDEFINSTR)) ;
}
```

Code layout

Missing exceptions code file

```
<<missing_exceptions.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Exception handler for unspecified exception vectors.
 *--
 */

/*
 * Include files
 */
#include <stdio.h>
#include <stddef.h>
#include <inttypes.h>
#include <string.h>
#include "useful.h"
<<missing exceptions: include files>>
/*
 * Constants
 */
<<missing exceptions: constants>>
/*
 * Data Type Declarations
 */
```

```
<<missing exceptions: data type declarations>>
/*
 * Forward References
 */
<<missing exceptions: forward references>>
/*
 * Static Data
 */
<<missing exceptions: static data>>
/*
 * Static Functions
 */
<<missing exceptions: static functions>>
/*
 * External Functions
 */
<<missing exceptions: external functions>>
```

Newlib Support

The newlib library defines a set of "system calls" which define a basic set of functions upon which it depends. The version of newlib used here stubs out most of those calls. System features such as the `assert` macro ultimately end up invoking `_exit`. To gain control over the default implementation supplied with the library, it is necessary to override the `_exit()` function. The solution here is simple. An `_exit` function is created which is linked in before the standard libraries to satisfy the symbol reference. The implementation simply invokes `sys_panic`.

```
<<_exit.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   _exit function replacement for the newlib version.
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
#include <stdio.h>
#include <unistd.h>
#include <stdarg.h>
#include <stdnoreturn.h>
#include "sys_svc_req.h"
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
#ifdef  USE_UART
#   include "dev_svc_uart_req.h"
#endif /*  USE_UART */
/*
 * External Functions
 */
noreturn void
exit(
```

```
    int status)
{
    _exit(status) ;
}

noreturn void
_exit(
    int status)
{
    sys_panic("_exit called with %d status", status) ;
}

noreturn void
abort(void)
{
#       ifdef USE_SEGGER_RTT

    while (SEGGER_RTT_HasDataUp(0) != 0) {
        ; // empty
    }

#       endif /*  USE_SEGGER_RTT */

    sys_panic("aborted") ;
}

#if defined(USE_SEGGER_RTT)

#include "SEGGER_RTT.h"

int
_write(
    int file,
    void const *ptr,
    size_t len)
{
    (void)file ;                        // not used
    SEGGER_RTT_Write(0, ptr, len);
    return (int)len;
}

__attribute__((format(printf, 1, 2)))
int
printf(
    char const *format,
    ...)
{
    char buf[256] ;

    va_list ap ;
    va_start(ap, format) ;
    int len = vsnprintf(buf, sizeof(buf), format, ap) ;
    va_end(ap) ;

    return _write(0, buf, len) ;
}

#endif /* USE_SEGGER_RTT */
```

Summary

In this chapter, several functions used to catch unanticipated behavior have been provided. The `SystemMissingException()` function serves this role for exception processing and the `panic()` and `sys_panic()` functions provide similar abilities to software. To support the newlib "C" library, an implementation of `_exit()` is also included to insure that all fatal errors from the standard library end up creating a panic condition and are caught. Invoking any of these functions is taken as non-recoverable and ultimately causes an abnormal termination. Their value lies in storing away the circumstances of the situation in memory that is *not* initialized upon routine resets.

Note this subject area is not entirely complete. No means have been provided to retrieve the data exception fault status. For the time being it is necessary to have a debugger or a connected terminal. This area is revisited later to complete the functionality needed for post-mortem examination of abnormal system termination.

Chapter 6

Batten Down the Hatches

In the second chapter of this book, a significant system design decision to separate privileged from unprivileged execution was put in place. Some of the implications of that decision were shown when the foreground/background request/notification scheme was built. This chapter shows how execution is further constrained by using the Memory Protection Unit (MPU).

Partitioning Memory Usage

The Background Notification Queue concept should not be conflated with the concept of execution privilege. The queue would be necessary even if privileged and unprivileged execution had not been separated. An IRQ handler must respond to the environment and queuing a notification allows the system to track the environment while other work is ongoing. The separation between privileged and unprivileged execution forced the use of the `SVC` instruction to make requests to the foreground. The `SVC` instruction is the mechanism specifically designed into the processor architecture to support obtaining privileged services.

As part of the foreground/background interactions, care was taken to insure that all data moved from the foreground into the background was done in an unprivileged manner. Specially designed processor instructions were used to insure that the background could not *trick* the foreground into writing into memory to which it otherwise should have no access.

It is also fitting to reiterate that the continuing efforts to constrain the application execution are *not* directed primarily to thwart malicious coding or to make the system suitable for executing application code of arbitrary provenance. The efforts are directed at early detection of software bugs and creating an environment where the undiscovered bugs in a deployed system are less likely to cause damage and are more easily diagnosed.

Until now, the mitigation efforts have used processor architectural features to constrain execution and trap any faults raised by the processor when it detects that software is operating the core improperly. The mitigation given by processing privilege alone is not enough to keep background execution constrained in terms of its memory accesses. Now, attention turns to how memory is used and to design a scheme where unprivileged execution is denied access to privileged memory areas.

Types of Memory Usage

It would be most convenient if all memory were the same and it was not necessary to make up special rules for memory usage. As a first order approximation, addresses are just addresses. Loads and stores from and to memory are not generally concerned with all the different characteristics that the underlying memory can have.

Upon closer examination, the following types of values are placed in processor memory.

Instructions
: All modern computers store their program in memory and the processor fetches and executes instructions from that memory[1].

[1] Historically, this is not the case. ENIAC was first programmed external to its memory, which was used only to hold data. Later improvements added the ability to store a program.

Data
Values manipulated by the processor are stored in its memory.

Constants
Values which don't change during the running of an application are stored in memory which does not necessarily allow writing.

Peripheral access
Memory also serves as the communications interface between the processor core and its peripheral devices.

Ignoring run time code generation and abominations such as self-modifying code, memory devoted to instructions is only ever read as part of an instruction fetch by the processor. In a microcontroller system where there typically is no external storage, memory holding instructions must survive power cycles. Modern microcontroller chips usually use some form of programmable memory which survives power cycles and which can be erased and re-programmed, *e.g.* flash memory.

Some data values are only ever read by the program, while others are updated by the program execution. This distinction between read only and read/write data is primarily driven by the economics of the electronics, namely there is significantly more read only memory available in a typical microcontroller than there is read/write memory. Read/write memory is typically implemented as static RAM.

Assuming processor instructions are not generated during run time, memory can be further restricted so that no execution from memory devoted to data can occur. This is *not* a general rule for all systems. There are rarer situations where code is placed in RAM for execution. For example, on some microcontroller chips, programming flash memory at run time requires executing the programming code from RAM since you would otherwise be erasing and controlling the same memory area from which instructions are fetched. Some flash memory controllers in chips cannot support instruction fetch and programming access concurrently.

Modern microcontroller chips map peripheral device control to memory locations. Older microprocessor designs sometimes used a processor I/O bus for device access. This too was driven by the economics of the situation. Older designs did not include built-in peripheral devices, and simplified I/O bus designs required less circuitry to integrate external devices to the system. Those circumstances no longer apply and devices present an interface which is mapped into the processor address space as "registers."

With these considerations, a first partitioning of memory is:

- Read only memory for instructions.
- Read only memory for constant data.
- Read/write memory for mutable data.
- Peripheral device access.

Consideration for separating privileged and unprivileged execution leads to further partitioning of memory usage. It does not exactly double the categories since all peripheral device access is only suitable for privileged execution. Considering two different execution privileges, a suitable partitioning of memory would be:

- Privileged read only memory for instructions.
- Privileged read only memory for constant data.
- Privileged read/write memory for mutable data.
- Privileged peripheral device access.
- Unprivileged read only memory for instructions.
- Unprivileged read only memory for constant data.
- Unprivileged read/write memory for mutable data.

The following figure shows the layout of the memory partitioning.

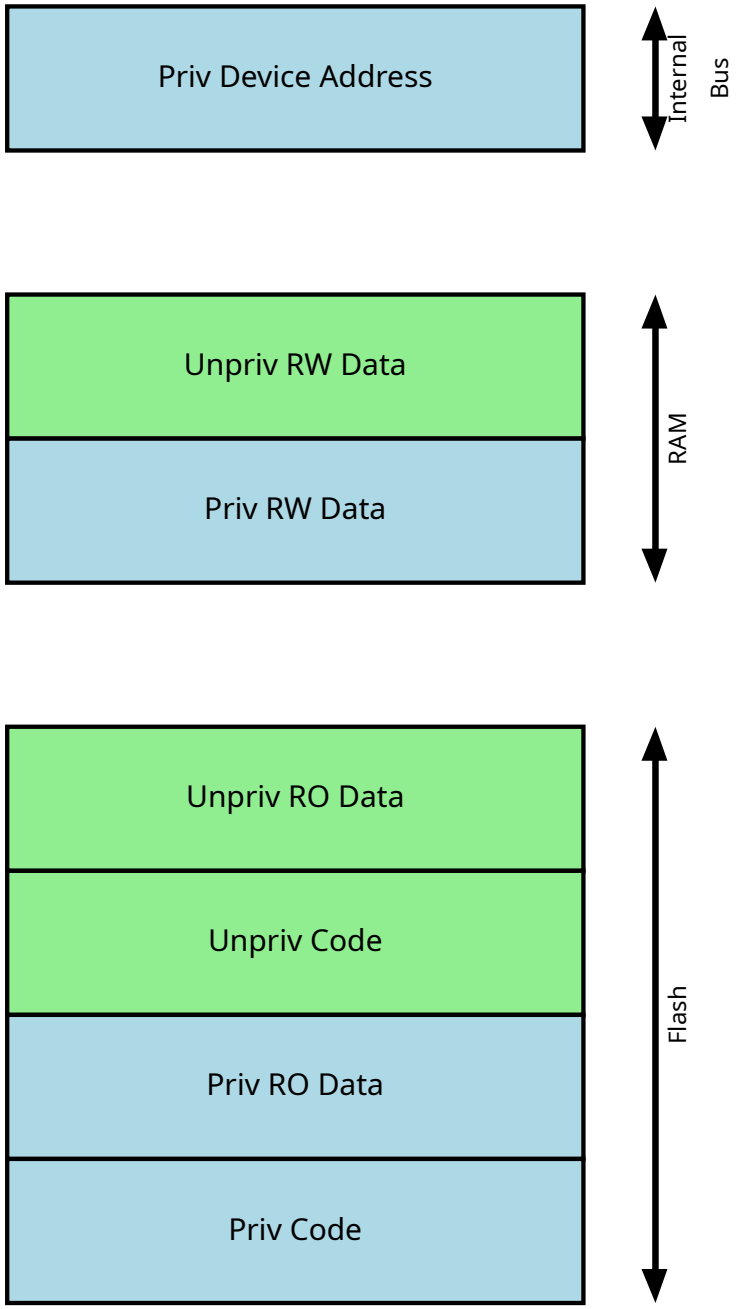

Figure 6.1: Memory Usage Partitioning

Enforcing Memory Usage

Given the goals presented in the previous section, the design scheme to enforce the separation of code and data between privileged and unprivileged execution involves:

- Programming the Memory Protection Unit (MPU) of the Cortex-M4 core to enforce the separation scheme.
- Separating the privileged from the unprivileged code/data so they may be placed in different memory areas.

- Constructing a linker script to place the code and data in the proper memory locations.

The details of programming the MPU are not discussed here. Code is presented to accomplish the MPU programming later. There are many good references as to what the MPU does and how it is controlled. The functions provided by CMSIS are used to get the register and bit fields correct. The focus in this section is on those controls of the MPU which affect how the designed memory usage scheme is enforced.

MPU Control Parameters

There are three characteristics about the operation of the MPU which are of concern.

Memory region
A memory region is the MPU concept of how portions of memory are treated. The MPU defines eight memory regions. Memory regions have a size which can range from 32 bytes to 4 GiB. For a given region size, the memory address for the region *must* be on an address boundary which is aligned to the size. This manner in which the MPU operates constrains allocation of memory regions and their placement by the linker and implies that larger regions have fewer places in memory where they can be placed. Memory regions also have a priority, where the highest numbered region takes precedence if multiple regions overlap a given portion of memory. This design does not use overlapping memory regions, so region numbering is somewhat arbitrary.

Access permissions
The access permissions on a region have the property that if any unprivileged access is allowed to a region, some form of privileged access is also granted. A region may prohibit unprivileged access, but the region remains accessible by privileged code. For this design, memory regions intended for unprivileged access, are granted the same permissions to privileged access. For memory regions intended for privileged access, no unprivileged access is granted.

Disallowing execution
Whether or not instructions may be fetched from a region is an independent attribute of an MPU region. In this design, all execution is disallowed except for memory regions specifically designated to hold instructions.

The following table shows how the design requirements for memory protection are mapped onto MPU regions and gives the needed MPU access permissions to implement the protection scheme.

Table 6.1: MPU Region Usage

Region	Usage	Privileged Access	Unprivileged Access
0	Privileged Code	Read Only	None
1	Privileged RO Data	Read Only, No Execute	None
2	Privileged RW Data	Read/Write, No Execute	None
3	Peripheral Devices	Read/Write, No Execute	None
4	Unprivileged Code	Read Only	Read Only
5	Unprivileged RO Data	Read Only, No Execute	Read Only, No Execute
6	Unprivileged RW Data	Read/Write, No Execute	Read/Write, No Execute

There remain two tasks to consider before the necessary code to program the MPU can be designed.

1. Privileged code/data must be separated from unprivileged.
2. A linker script must be written to place the previously characterized memory sections into the physical memory provided by the SOC.

Tweezing Apart Privilege

The strategy for separating privileged code/data from unprivileged is to use incremental linking. Incremental linking is a technique supported by the linker to combine several object files to produce a new object file which may be used in another, subsequent linking process. All code files which contain functions and any associated data which execute in handler mode are incrementally linked into a single object file. The resulting incrementally-linked object file still contains the necessary relocation information and it serves as the source of code and data that is placed in the privileged memory areas. This file is named, `privileged.o`, and is used in the linker script, shown below, as the source of `text`, `rodata` and `bss` for the privileged memory areas.

To obtain the unprivileged code/data, all the remaining files or libraries are included in the link and the linker is directed to exclude anything contained in `privleged.o` from the sections where the unprivileged code and data are placed.

There are a few consequences to note for this scheme.

- Only code intended for handler mode operations is to be included in `privileged.o`. This includes startup code, exception handling code and the system and device realm code from a previous chapter.
- Common code, such as standard library code or shared utility functions, that is used by both privileged and unprivileged execution is *not* included in `privileged.o`, but rather it is placed in unprivileged memory. This is acceptable for present purposes. However, if the intent is to run untrusted application code, then an alternative strategy would be to link `privileged.o` as a stand-alone program with no external references (except to `main`). Such an arrangement would duplicate any common code used by both privileged and unprivileged execution, but would create better separation. This is analogous to what an operating system does, but in this situation comes with the cost of increased memory usage.
- The external definitions in `privileged.o` are still visible during link time, but unprivileged code/data must be disallowed from making any cross references to symbols in `privileged.o`. Fortunately, the linker supports commands which check such cross references and cause a link-time error if any unprivileged linker section makes a reference to a symbol defined in a privileged section.

Note that having the linker check for unintended cross references between unprivileged and privileged memory areas does not detect unprivileged access to peripheral devices. Peripheral devices are not typically accessed using program symbols and do not appear as a memory block in the linker script. Rather, physical addresses in the form of pre-processor symbols are cast to be pointers to data structures which are contrived to be "shaped" like the register layout of the device. Thus, there are no program symbols corresponding to the peripheral registers and the linker has no knowledge of the peripheral device memory. However, the MPU detects such disallowed accesses at run time.

Linker Script

The linker script presented here is quite different from the one used in Chapter 1. That linker script was not concerned about execution privilege and treated memory as either flash or RAM. The only complications in the script arose from our defining memory regions which survive reset.

The linker script presented here has the same general layout, but with some important differences pointed out below.

```
<<apollo3_priv.ld>>=
<<copyright info>>
<<apollo3 priv: entry point>>
<<apollo3 priv: memory layout>>
<<apollo3 priv: cross reference checks>>
<<apollo3 priv: section descriptions>>
```

Entry point

The entry point remains the same, namely the `Reset_Handler`.

```
<<apollo3 priv: entry point>>=
ENTRY(Reset_Handler)
```

Memory regions

As mentioned previously, MPU regions must be aligned to an address which is a multiple of their length, *e.g.* a 64K code section must be aligned to a 64K address boundary. This complicates the placement of memory regions in the flash and RAM of the SOC. When region alignment was not a concern, it did not matter into which parts of flash or RAM the various bits of code and data were placed. Now it matters greatly.

The chosen mechanism to solve the alignment of allocated memory to MPU region addresses is to define, as part of the memory specified to the linker, a separate block of memory with its own location and size. It is not an ideal solution. All the allocation calculations must be done by hand. *A priori* it is difficult to know how much memory should be allocated to a particular usage. However, managing the sections manually makes clear any holes in our memory map and the linker emits an error if any of the defined memory boundaries overflows.

```
<<apollo3 priv: memory layout>>=
MEMORY
{
  priv_text (rx) : ORIGIN = 0x00010000, LENGTH = 64K
  priv_rodata (r) : ORIGIN = ORIGIN(priv_text) + LENGTH(priv_text), LENGTH = 64K
  unpriv_rodata (r) : ORIGIN = ORIGIN(priv_rodata) + LENGTH(priv_rodata), LENGTH = 64K
  unpriv_text (rx) : ORIGIN = ORIGIN(unpriv_rodata) + LENGTH(unpriv_rodata), LENGTH = 128K

  priv_rwdata (rw) : ORIGIN = 0x10000000, LENGTH = 64K
  unpriv_rwdata (rw) : ORIGIN = ORIGIN(priv_rwdata) + LENGTH(priv_rwdata), LENGTH = 64K
}
```

This simple scheme allocates 64K memory blocks to each type of memory except for unprivileged code which is given 128K (note peripheral device access has a defined memory location and is not part of the program linkage). The expressions giving the origin address of the memory blocks insures they are placed sequentially in either flash or RAM memory.

Prohibiting cross references

The linker `NOCROSSREFS_TO` command is used to request the linker to insure no symbol cross references occur from any of the unprivileged memory blocks to privileged ones. The command can be used to give link-time errors if unprivileged code tries to reference directly any privileged memory. Recall, that all access to privileged execution happens via the `SVC` interface which is a specific processor instruction and *not* a function with an external symbol.

```
<<apollo3 priv: cross reference checks>>=
NOCROSSREFS_TO(.priv_text .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_TO(.priv_rodata .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_TO(.priv_data .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_TO(.main_stack .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_TO(.noinit_data .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_TO(.priv_bss .unpriv_text .unpriv_rodata .unpriv_rwdata)
NOCROSSREFS_TO(.noinit_bss .unpriv_text .unpriv_rodata .unpriv_rwdata)
```

Defining the linker sections

For each memory block, whose access is controlled with the MPU, has a section definition to describe the content of the block. Note that the names for linker sections are the same as the name of the memory block, adding the conventional leading period. This is just a convention meant to help keep track of what is going where.

When defining the contents of the memory blocks, linker generated are used to generate symbols giving the starting address and length of the various memory blocks. These linker symbols are used to configure the MPU. This gives a single location, namely the linker script, where the location and usage of memory is defined.

There are a separate set of linker script commands for each memory block defined above. Note, there are no linker commands associated with the memory for peripheral devices. Peripheral devices addresses are only known by *dead reckoning*, *i.e.* by pre-processor defined constants.

```
<<apollo3 priv: section descriptions>>=
SECTIONS
{
    <<priv text section>>
    <<priv rodata section>>
    <<unpriv text section>>
    <<unpriv rodata section>>
    <<priv ram sections>>
    <<unpriv ram sections>>
    <<mpu region symbols>>

    PROVIDE(end = .) ;
}
```

As in the previous linker script, the exception vector table must be placed at the first locations of memory loaded. The use of the `*privileged.o` qualifier indicates that the section must come from the object file in which has been incrementally linked with all the privileged code and data.

```
<<priv text section>>=
.priv_text :
{
    KEEP(*privileged.o(.vectors*))
    KEEP(*privileged.o(.ble_patch*))

    . = ALIGN(4) ;
    *privileged.o(.text*)

    . = ALIGN(4) ;
    PROVIDE(__priv_text_end__ = .) ;
} > priv_text
```

Following the same pattern, the privileged read only data is loaded. Additionally, the various copy tables used to initialize RAM at start up time are created. Recall that the code used to copy initializers to RAM supports multiple entries in order to place the initializers into multiple segments of RAM. This is not by accident.

```
<<priv rodata section>>=
.priv_rodata :
{
    *privileged.o(.rodata*)

    <<memory initialization tables>>

    PROVIDE(__priv_rodata_end__ = .) ;
} > priv_rodata
```

The definitions of the copy and zero table descriptors follows the same conventions as our previous linker script. The only difference is the addition of multiple segments to accommodate the restrictions on access to the memory.

```
<<memory initialization tables>>=
. = ALIGN(4) ;

 __data_copy_table_start__ = . ;
LONG(LOADADDR(.priv_data))
LONG(ADDR(.priv_data))
LONG(SIZEOF(.priv_data) / 4)

LONG(LOADADDR(.unpriv_data))
LONG(ADDR(.unpriv_data))
LONG(SIZEOF(.unpriv_data) / 4)
__data_copy_table_end__ = . ;
```

```
__zero_table_start__ = . ;
LONG(ADDR(.priv_bss))
LONG(SIZEOF(.priv_bss) / 4)

LONG(ADDR(.unpriv_bss))
LONG(SIZEOF(.unpriv_bss) / 4)
__zero_table_end__ = . ;

 __stack_table_start__ = . ;
LONG(ADDR(.process_stack))
LONG(SIZEOF(.process_stack) / 4)

LONG(ADDR(.main_stack))
LONG(SIZEOF(.main_stack) / 4)
__stack_table_end__ = . ;

 __noinit_data_copy_table_start__ = . ;
LONG(LOADADDR(.noinit_data))
LONG(ADDR(.noinit_data))
LONG(SIZEOF(.noinit_data) / 4)
__noinit_data_copy_table_end__ = . ;

__noinit_bss_table_start__ = . ;
LONG(ADDR(.noinit_bss))
LONG(SIZEOF(.noinit_bss) / 4)
__noinit_bss_table_end__ = . ;
```

Unprivileged code and data continues to be loaded in the previous manner. Note here the `EXCLUDE_FILE(*privileged.o)` construct insures that no privileged code or data is placed in the unprivileged memory areas.

```
<<unpriv text section>>=
.unpriv_text :
{
    EXCLUDE_FILE(*privileged.o)*(.text*)

    . = ALIGN(4) ;
    PROVIDE(__unpriv_text_end__ = .) ;
} > unpriv_text
```

```
<<unpriv rodata section>>=
.unpriv_rodata :
{
    EXCLUDE_FILE(*privileged.o)*(.rodata*)
} > unpriv_rodata
```

Access to the Build ID is allowed as unprivileged. Application code should be able to report the Build ID without having to resort to privileged execution.

```
<<unpriv rodata section>>=
.build_id : ALIGN(4)
{
    PROVIDE(SystemBuildIDNote = .) ;
    *(.note.gnu.build-id)
} > unpriv_rodata
```

Likewise, exception unwinding information access is not privileged, especially since there is no anticipation having any of it.

```
<<unpriv rodata section>>=
.ARM.extab : ALIGN(4)
{
    *(.ARM.extab* .gnu.linkonce.armextab.*)
```

```
} > priv_rodata

. = ALIGN(4) ;
PROVIDE(__exidx_start = .) ;

.ARM.exidx : ALIGN(4)
{
    *(.ARM.exidx* .gnu.linkonce.armexidx.*)
} > priv_text
PROVIDE(__exidx_end = .) ;

. = ALIGN(4) ;
PROVIDE(__unpriv_rodata_end__ = .) ;
```

Privileged RAM consists of not only the data and bss sections for the privileged code, but also of the main stack and those memory areas which survive reset. This implies that access to the contents of memory which survives reset is intended to go through privileged code.

```
<<priv ram sections>>=
.main_stack (NOLOAD) : ALIGN(8)
{
    PROVIDE(__main_stack_limit__ = .) ;

    KEEP(*(.main_stack*))

    . = ALIGN(8) ;
    PROVIDE(__main_stack_top__ = .) ;
} > priv_rwdata

.priv_data : AT (LOADADDR(.priv_rodata) + SIZEOF(.priv_rodata)) ALIGN(4)
{
    PROVIDE(__priv_data_start__ = .) ;

    *privileged.o(.data*)

    . = ALIGN(4) ;
    PROVIDE(__priv_data_end__ = .) ;
} > priv_rwdata

.noinit_data : AT(LOADADDR(.priv_data) + SIZEOF(.priv_data)) ALIGN(4)
{
    PROVIDE(__noinit_data_start__ = .) ;

    *(.noinit_data*)

    . = ALIGN(4) ;
    PROVIDE(__noinit_data_end__ = .) ;
} > priv_rwdata

.priv_bss (NOLOAD) : ALIGN(4)
{
    PROVIDE(__priv_bss_start__ = .) ;

    *privileged.o(.bss*) ;
    *privileged.o(COMMON) ;

    . = ALIGN(4) ;
    PROVIDE(__priv_bss_end__ = .) ;
} > priv_rwdata

.noinit_bss (NOLOAD) :
{
```

```
    PROVIDE(__noinit_bss_start__ = .) ;
    PROVIDE(noinit_status = .) ;
    . += 8 ;                    /* 8 == sizeof(NoinitSegmentStatus) */

    *(.noinit_bss*)

    . = ALIGN(4) ;
    PROVIDE(__noinit_bss_end__ = .) ;
} > priv_rwdata
```

Unprivileged RAM access is restricted to the data and bss segments for unprivileged code and the process stack used by unprivileged code.

```
<<unpriv ram sections>>=
.process_stack (NOLOAD) : ALIGN(8)
{
    __process_stack_limit__ = . ;

    KEEP(*(.process_stack*))

    . = ALIGN(8) ;
    __process_stack_top__ = . ;
} > unpriv_rwdata

.unpriv_data : AT (__unpriv_rodata_end__) ALIGN(4)
{
    PROVIDE(__unpriv_data_start__ = .) ;

    EXCLUDE_FILE(*privileged.o)*(.data*)

    . = ALIGN(4) ;
    PROVIDE(__unpriv_data_end__ = .) ;
} > unpriv_rwdata

.unpriv_bss (NOLOAD) : ALIGN(4)
{
    PROVIDE(__unpriv_bss_start__ = .) ;

    EXCLUDE_FILE(*privileged.o)*(.bss*)
    EXCLUDE_FILE(*privileged.o)*(COMMON)

    . = ALIGN(4) ;
    PROVIDE(__unpriv_bss_end__ = .) ;
} > unpriv_rwdata
```

Global symbols provided by the linker are used to configure the MPU. The values of these symbols are used directly for the MPU configuration and to accomplish that the linker must do some computations. Recall that the MPU requires knowledge of the size of a memory region to which protections are applied. The region size is encoded in a particular way since it is actually a bit field in a larger register context. Hardware encodings are usually made for the convenience of hardware operation. The encoding of region size is a small unsigned integer number which represents the number of bits in a memory address which are ignored when considering whether a given address is controlled by an MPU region **minus 1**. For example, a 32 Byte MPU memory region size is specified as 4. Aligning to a 32 Byte boundary means that the lower 5 bits of an address are discarded. Similarly, a 256 Byte region size is specified as 7 since the 256 Byte boundaries are determined by ignoring the low order 8 bits. Mathematically, the region size is encoded as:

$$size_{encoded} = \lceil \log_2(size_{region}) \rceil - 1$$

Conveniently, the linker has a command which computes $\lceil \log_2 \rceil$ so global symbols can be created for the MPU region size which is encoded in such a way that the symbol value may be directly used to configure the MPU. This usage is, admittedly, obscure, but has the advantage of keeping all the MPU region specification in a single place, namely the linker script.

```
<<mpu region symbols>>=
```

```
__priv_text_start__ = ORIGIN(priv_text) ;
__priv_text_size__ = LOG2CEIL(LENGTH(priv_text)) - 1 ;

__priv_rodata_start__ = ORIGIN(priv_rodata) ;
__priv_rodata_size__ = LOG2CEIL(LENGTH(priv_rodata)) - 1 ;

__priv_rwdata_start__ = ORIGIN(priv_rwdata) ;
__priv_rwdata_size__ = LOG2CEIL(LENGTH(priv_rwdata)) - 1 ;

__unpriv_text_start__ = ORIGIN(unpriv_text) ;
__unpriv_text_size__ = LOG2CEIL(LENGTH(unpriv_text)) - 1 ;

__unpriv_rodata_start__ = ORIGIN(unpriv_rodata) ;
__unpriv_rodata_size__ = LOG2CEIL(LENGTH(unpriv_rodata)) - 1 ;

__unpriv_rwdata_start__ = ORIGIN(unpriv_rwdata) ;
__unpriv_rwdata_size__ = LOG2CEIL(LENGTH(unpriv_rwdata)) - 1 ;
```

To refer to the linker symbols in code, a set of external references are required. The data type given to these external declarations is of little consequence. When used in code to initialize the MPU, the symbol address is cast to the appropriate bit pattern.

```
<<mpu init: external declarations>>=
extern uint32_t __priv_text_start__ ;
extern uint32_t __priv_text_end__ ;
extern uint32_t __priv_text_size__ ;

extern uint32_t __priv_rodata_start__ ;
extern uint32_t __priv_rodata_end__ ;
extern uint32_t __priv_rodata_size__ ;

extern uint32_t __priv_rwdata_start__ ;
extern uint32_t __priv_rwdata_end__ ;
extern uint32_t __priv_rwdata_size__ ;

extern uint32_t __unpriv_text_start__ ;
extern uint32_t __unpriv_text_end__ ;
extern uint32_t __unpriv_text_size__ ;

extern uint32_t __unpriv_rodata_start__ ;
extern uint32_t __unpriv_rodata_end__ ;
extern uint32_t __unpriv_rodata_size__ ;

extern uint32_t __unpriv_rwdata_start__ ;
extern uint32_t __unpriv_rwdata_end__ ;
extern uint32_t __unpriv_rwdata_size__ ;
```

Configuring the MPU

The following table shows the memory attributes used for the seven defined memory regions. The values for TEX, Shareable, Cacheable, and Bufferable are taken from [defguide], Table 11.10, p364.

Table 6.2: MPU Memory Attributes

Region	TEX	Shareable	Cacheable	Bufferable
priv text	0	0	1	0
priv rodata	0	0	1	0
priv wrdata	0	1	1	0
peripherals	0	1	0	1

Table 6.2: (continued)

Region	TEX	Shareable	Cacheable	Bufferable
unpriv text	0	0	1	0
unpriv rodata	0	0	1	0
unpriv wrdata	0	1	1	0

The values for the MPU registers that implement the memory protection scheme can now be specified. The following data structure encodes the memory attributes into the register values using the CMSIS MPU functions.

```
<<mpu init: region configuration>>=
ARM_MPU_Region_t apollo3_mpu_regions[] = {
    {
        .RBAR = ARM_MPU_RBAR(0, (uint32_t)&__priv_text_start__),
        .RASR = ARM_MPU_RASR(0, ARM_MPU_AP_PRO,
                    0, 0, 1, 0, 0, (uint32_t)&__priv_text_size__),
    },  //          typeext share cache buff subreg size
    {
        .RBAR = ARM_MPU_RBAR(1, (uint32_t)&__priv_rodata_start__),
        .RASR = ARM_MPU_RASR(1, ARM_MPU_AP_PRO,
                    0, 0, 1, 0, 0, (uint32_t)&__priv_rodata_size__),
    },
    {
        .RBAR = ARM_MPU_RBAR(2, (uint32_t)&__priv_rwdata_start__),
        .RASR = ARM_MPU_RASR(1, ARM_MPU_AP_PRIV,
                    0, 1, 1, 0, 0, (uint32_t)&__priv_rwdata_size__),
    },
    {
        .RBAR = ARM_MPU_RBAR(3, UINT32_C(0x40000000)),
        .RASR = ARM_MPU_RASR(1, ARM_MPU_AP_PRIV,
                    0, 1, 0, 1, 0, ARM_MPU_REGION_SIZE_256MB),
    },
    {
        .RBAR = ARM_MPU_RBAR(4, (uint32_t)&__unpriv_text_start__),
        .RASR = ARM_MPU_RASR(0, ARM_MPU_AP_RO,
                    0, 0, 1, 0, 0, (uint32_t)&__unpriv_text_size__),
    },
    {
        .RBAR = ARM_MPU_RBAR(5, (uint32_t)&__unpriv_rodata_start__),
        .RASR = ARM_MPU_RASR(1, ARM_MPU_AP_RO,
                    0, 0, 1, 0, 0, (uint32_t)&__unpriv_rodata_size__),
    },
    {
        .RBAR = ARM_MPU_RBAR(6, (uint32_t)&__unpriv_rwdata_start__),
        .RASR = ARM_MPU_RASR(1, ARM_MPU_AP_FULL,
                    0, 1, 1, 0, 0, (uint32_t)&__unpriv_rwdata_size__),
    },
} ;
```

Initializing the MPU

The `SystemMemProtectInit` function is reimplemented and supplied as a strong symbol to override the default implementation. With the configuration information in a variable, a CMSIS function is used to load the configuration and enable the MPU.

```
<<mpu init: external functions>>=
void
```

```
SystemMemProtectInit(void)
{
    <<mpu init: region configuration>>

    ARM_MPU_Disable() ;
    ARM_MPU_Load(apollo3_mpu_regions, COUNTOF(apollo3_mpu_regions)) ;
    ARM_MPU_Enable(MPU_CTRL_PRIVDEFENA_Msk) ;                          // (1)
}
```

(1) Privileged code is allowed to access the default region map. This facilitates access to the system control block without having to define another MPU region.

Code Layout

```
<<system_mem_protect_init.c>>=
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Code to initialize the Cortex-M4 MPU
 *--
 */

/*
 * Include files
 */
#include <stdint.h>
#include "useful.h"
#include "apollo3.h"
/*
 * External Declarations
 */
<<mpu init: external declarations>>
/*
 * External Functions
 */
<<mpu init: external functions>>
```

Summary

This chapter shows how to configure the Cortex-M4 MPU to segregate memory usage between privileged and unprivileged execution. This design uses seven MPU memory regions to separate privileged access from unprivileged access across code, read-only data, and read-write data. The seventh region is used to restrict peripheral device access to privileged execution. The linker script was rewritten to layout memory on specific boundaries to accommodate the manner in which the MPU operates. Linker script symbols were defined whose values were directly used to configure the MPU registers.

Chapter 7

Time Mancery

Timing services required by reactive systems tend to fall into the following broad areas.

Generating waveforms
Generating a waveform, either analog or digital, to control external peripherals is a major function of many systems. Motor control and audio output are just a couple of examples. Many external peripheral devices use specifically patterned digital waveforms for communications, *e.g.* I2C, SPI, and UART. Complex waveform generation, especially to implement a defined protocol, usually requires specialized microcontroller peripherals. It is difficult to generate intricate waveforms using "bit banging" by the MCU, especially if the required speed is close to the clock rate of the processor. Modern SOC's have timer blocks which which can operate precisely at high speeds.

Measuring waveforms
Measuring analog and digital waveforms are important tasks that arise when interacting with the environment of the system. Precise measurement of pulse widths or precise sampling of an analog signal requires accurate timing, usually achieved with timer peripherals.

Internal software timing
When software deals with the external world, frequently, time is used to determine if a desired interaction will ever happen or needs to be retried. Communications protocol timeouts are an archetypical example, but waiting on external peripherals to accomplish some task is also common. This type of timing does not usually have the strict precision requirements as waveform timing.

Tracking human time
Humans keep time in an intricate way that is associated with astronomical phenomenon such as the rotational period of the Earth around the Sun. Specialized hardware for tracking human time is also usually available in the form of a so-called Real Time Clock (RTC).

Tracking computer time
Having a computer monitor its own performance is a special case of time measurement. For this case, a specific set of base time services is required that support characterization of the performance of the software execution.

This chapter presents designs for the later three cases. The topics of generating or measuring waveforms are deferred to later when additional capabilities to interface with the environment are described. The focus in this chapter is on keeping time by counting clock pulses.

Driving Software with Time

For most software purposes, initiating processing with approximately millisecond resolution works well. The frequency is not too high to require extraordinary considerations. The type of timing considered here is useful for timeouts, for example, in handling communications, low speed sensor sampling, or button debouncing. Since there are many distinct timing service requests from application software, the main concern is to make best use of the available timing peripherals. The number of timing requests

by application software is not generally predictable but one can anticipate that there are many more software timing requests than physical timers. Multiplexing physical timer peripherals is one solution. To avoid unproductive computation, the design for multiplexing a timer must *not* involve periodic execution which accomplishes little more than decrementing a counter. Hardware is available for that purpose. The intent is to have hardware peripherals precisely inform the software when some action is to be taken.

In this chapter two topics are covered.

1. First, a design concept for multiplexing a single physical timer to produce multiple time expiration notifications is presented.
2. Second, the code is shown for the device background requests and corresponding foreground proxies which, together, provide timing service to background software.

Timer Queue Concepts

Using a single timer peripheral to drive a timing queue stored in time-relative order is a particularly handy technique to multiplex timer use. A timer peripheral is used to signal the elapsed time for the queue element at the head of the queue. The subsequent elements in the queue record the amount of additional time, after the expiration of the previous entry, which must elapse before the element expires. When the element at the head of the timer queue expires, it is removed from the queue and the next element's expiration time is loaded into the timer. Thus, the total time elapsed before the second element expires is the expiration time for the first element plus the expiration time for the second one. This logic applies to all the following queue elements.

The following diagram illustrates these ideas by showing a schematic snapshot of a timer queue.

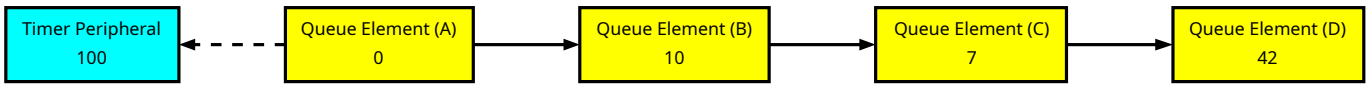

Figure 7.1: Snapshot of Timer Queue Operations

In this diagram, the yellow boxes represent timer queue elements and contain their relative counts. The cyan box represents the timer peripheral which is currently timing a 100 tick delay. This is the offset from the current time to when **Queue Element (A)** expires (note the count for **(A)** is zero indicating that no additional time beyond that in the timer peripheral is required). **Queue Element (B)** is to expire 10 ticks after **Queue Element (A)**. **(C)** is to expire 7 ticks after **(B)** and **(D)** is to expire 42 ticks after **(C)**. So, **Queue Element (D)** expires 100 + 0 + 10 + 7 + 42 = 159 ticks after the moment shown in the diagram.

Specifically, the design concept is:

- A queue of relative expiration times is maintained.
- A timer peripheral is devoted to counting down the expiration time of the element at the head of the queue.
- When the head element expires, a background notification is sent for the that element. A separate notification is also sent for all elements following the head element which have an additional expiration time of zero (*i.e.* those elements which were intended to expire at the same time as the head element).
- Each queue element also carries a *reload* time to support periodic expiration without having to process additional requests. A reload value of zero, indicates that the element produces only a single expiration notification.

The fundamental operations on the timer queue data structures are:

a. *insert* an element

b. *remove* an element

c. determine the *remaining* time before an element expires

It is also necessary to supply a request to "open" and "close" the timer queue so that a background notification proxy function may be specified. The background notification proxy implements the actions taken by background processing when a timer queue element expires.

Keeping the expiration times in the queue ordered by relative amounts of additional time that must elapse before the element expires minimizes the computation required to load the next time value into the timer peripheral. The next value to place into the timer peripheral is *pre-computed* and it is only necessary to walk the timer queue to expire elements. Upon finding the first non-zero expiration time, the timer is loaded with the value that counts the additional time before, what is now the head of the timer queue, expires. This is an attempt to minimize the timing *jitter* which is a consequence of using a single timer peripheral and requiring software to reload and control the peripheral.

Minimizing the computation required when a queue element expires comes at a cost. This scheme trades off lower costs at expiration for additional costs when an element is inserted or removed from the timer queue. In the insert case, the queue must be searched, in order, to find the appropriate place to insert the element accounting for the time that will have elapsed before the newly inserted element progresses to the head of the queue. When an element is removed, its relative elapsed time must be added to the next element in the queue to keep the timing correct for those elements which expire after the removed one. An update operation is also supported. Updating an element is equivalent to removal and re-insertion of the same element but with a new expiration time.

The next section shows the implementation of a timer queue and the basic operations that act upon it. Later, the timer queue operations are packaged into the background requests and foreground proxies according to the established pattern.

Timer Queue Elements

A timer queue element is the data object held in the timer queue. There are a couple of design decisions to be made with respect to the structure of the queue elements:

- A circular, doubly-linked list, which is treated as a queue, is used to hold the timer queue. Two link pointers make inserting and removing at arbitrary places in the queue much simpler. The list is circularly linked so that it appears empty when the *next* and *previous* pointers reference the list head itself.
- The timer queue linked list is *intrusive*, *i.e.* the structure of each queue element contains the storage necessary for the linked list pointers. This choice has two consequences:
    a. Managing the storage is simplified. It is not necessary to manage storage for separate links and element contents.
    b. Since the link is buried in a structure along with other queue element data, a `CONTAINER_OF` pre-processor macro is used to map a pointer to a structure element (in this case the linked list pointers which are needed to manage the queue) to a pointer to the containing structure (in this case the structure of the timer queue element which is used to contain specific timer attributes). You could place list linkage as the first member of the queue element structure. That would allow a cast from the queue element structure to the linked list structure. In this case, the design has opted for dubious pointer arithmetic which is hidden in a macro over dubious type casting which depends upon a side agreement for the placement of a structure member. Neither solution is ideal as both involve enough type casting to insure the compiler cannot aid in any coding error detection. This is just a case where awareness along with careful code and review is required.

The elements of the timer queue have the following structure.

```
<<dev realm timq proxy: data type declarations>>=
typedef struct {
    CLL_Links links ;
    TIMQ_TimeTicks expire_time ;
    TIMQ_TimeTicks reload_time ;
    SVC_DevNotifyClosure notify_closure ;
} TIMQ_Element ;
```

CLL_Links
: Linked list pointers to allow variables of type `TIMQ_Element` to be held in a linked list.

expire_time
: The minimum number of timer ticks which are to elapse before a notification is sent.

reload_time
: The number of timer ticks for repeated timer expiration. A `reload` value of 0, implies the timer element is used as a one-shot notification.

notify_closure
: A value returned in any background notification associated with the timer element.

The interface definitions for a circularly linked list are contained in a header file. One convenience offered by circular linking is to handle the boundary conditions at the head or tail of the queue. However, a *list head* object must be allocated, which in this case is just a variable of type `CCL_Links`.

```
<<dev realm timq proxy: include files>>=
#include "cl_list.h"
```

Since the timer queue is a list of queue elements, storage for the elements that are held in the queue must be allocated. This is one of the cases where is seems advisable to provide storage space and allocation as part of the foreground processing. You could design the interface such that queue element storage was managed by the background requests, but this would complicate the background request interface considerably and simply move the consideration of how to size the storage pool for queue elements to another part of the system. Allocating queue elements in the background would also expose essential control values to unprivileged processing.

The general purpose block allocator is used for the allocation and bookkeeping of the queue elements.

```
<<dev realm timq proxy: include files>>=
#include "block_alloc.h"
```

Sizing the storage pool for queue elements is problematic at this stage. As usual, a best guess is used with the understanding that it may need to be resized when a specific application is deployed.

```
<<dev realm timq proxy: constants>>=
#ifndef TIMQ_ELEMENT_POOL_SIZE
#   define  TIMQ_ELEMENT_POOL_SIZE 32
#endif /* TIMQ_ELEMENT_POOL_SIZE */
```

Separate storage pools are kept for each timer queue instance. You could consider having a single element pool which is shared among all timer queue instances. Unfortunately, that design would suffer from non-deterministic behavior, *i.e.* the number of elements available to be queued would depend upon how the pool of queue elements is used by other parts of the system. Deterministic behavior is preferred even at the cost of poorer utilization for some memory. Also, a single storage pool would mean that operations on the pool would need to disable multiple interrupts. The system timer peripheral has distinct IRQ vectors for each of the compare registers used to expire timer elements.

Two instances of the timer queue logical device are supported.

```
<<dev realm timq param: constants>>=
#define TIMQ_INSTANCES   2
```

The block allocation code gives a pre-processor macro to use for defining the allocator and its storage pool.

```
<<dev realm timq proxy: static data definitions>>=
DEFINE_BLOCK_ALLOCATOR(inst0_allocator, TIMQ_Element, TIMQ_ELEMENT_POOL_SIZE) ;
DEFINE_BLOCK_ALLOCATOR(inst1_allocator, TIMQ_Element, TIMQ_ELEMENT_POOL_SIZE) ;
```

Background Notification

Each queue element is given a small, zero-based, sequential integer value that is an identifier for the element. The element identifier is only unique with in the context of a given timer queue instance. Background processing is expected to use these identifiers when requesting timer queue element operations.

```
<<dev realm timq param: data type declarations>>=
typedef int16_t TIMQ_ElementID ;
```

A signed data type is chosen so that negative numbers can be used to indicate an invalid element ID. As discussed below, the values of type `TIMQ_ElementID` are actually the array indices into the storage pool from which the element was allocated. This is a simple and computationally cheap scheme which avoids using a pointer value as an identifier (don't want to expose privileged memory addresses to unprivileged background processing) or having to construct some other mapping between the queue element and an external identifier.

A separate notification is sent for each queue element. The notification denotes changes in status for the element. The status changes supported are:

a. The timer element has expired, *i.e.* at least the amount of time requested when the element was inserted has elapsed.

b. The timer element has expired and was reloaded to expire again in the future. This is the case when the timer element has a non-zero reload time.

c. The timer element was removed as a side effect of closing the timer queue instance. This notification insures that queue elements do not mysteriously disappear with no indication.

```
<<dev realm timq param: data type declarations>>=
typedef enum {
    timq_expired = 5,                   // ❶
    timq_reloaded,
    timq_discarded,
} TIMQ_NotifyStatus ;
```

❶ A lot of zeros float around in code. Start at something which has a lower probability of being a garbage memory value. This encoding also avoids some of the implicit "truthy-ness" which, in "C", can arise when integer values are coerced in boolean expressions.

The information carried in the background notification both identifies the timer queue element and gives a status value associated to the queue element.

```
<<dev realm timq param: data type declarations>>=
DECLARE_DEV_NOTIFICATION(SVC_DevTimqNotification,
    TIMQ_ElementID element_id ;
    TIMQ_NotifyStatus status ;
) ;
```

The background notification proxy takes the device specific form of the notification as a argument.

```
<<dev realm timq param: data type declarations>>=
typedef void (*SVC_DevTimqNotifyProxy)(SVC_DevTimqNotification const *const params) ;
```

System Timer Comparator Control

The code used for hardware control is compartmentalized, but the logic of the operation of the system timer peripheral hardware cannot be hidden. Here the essential rules of operation for the timer peripheral are discussed. The Apollo 3 SOC data sheet has the specifics. The following figure is taken from the Apollo 3 data sheet and shows a block diagram of the system timer peripheral.

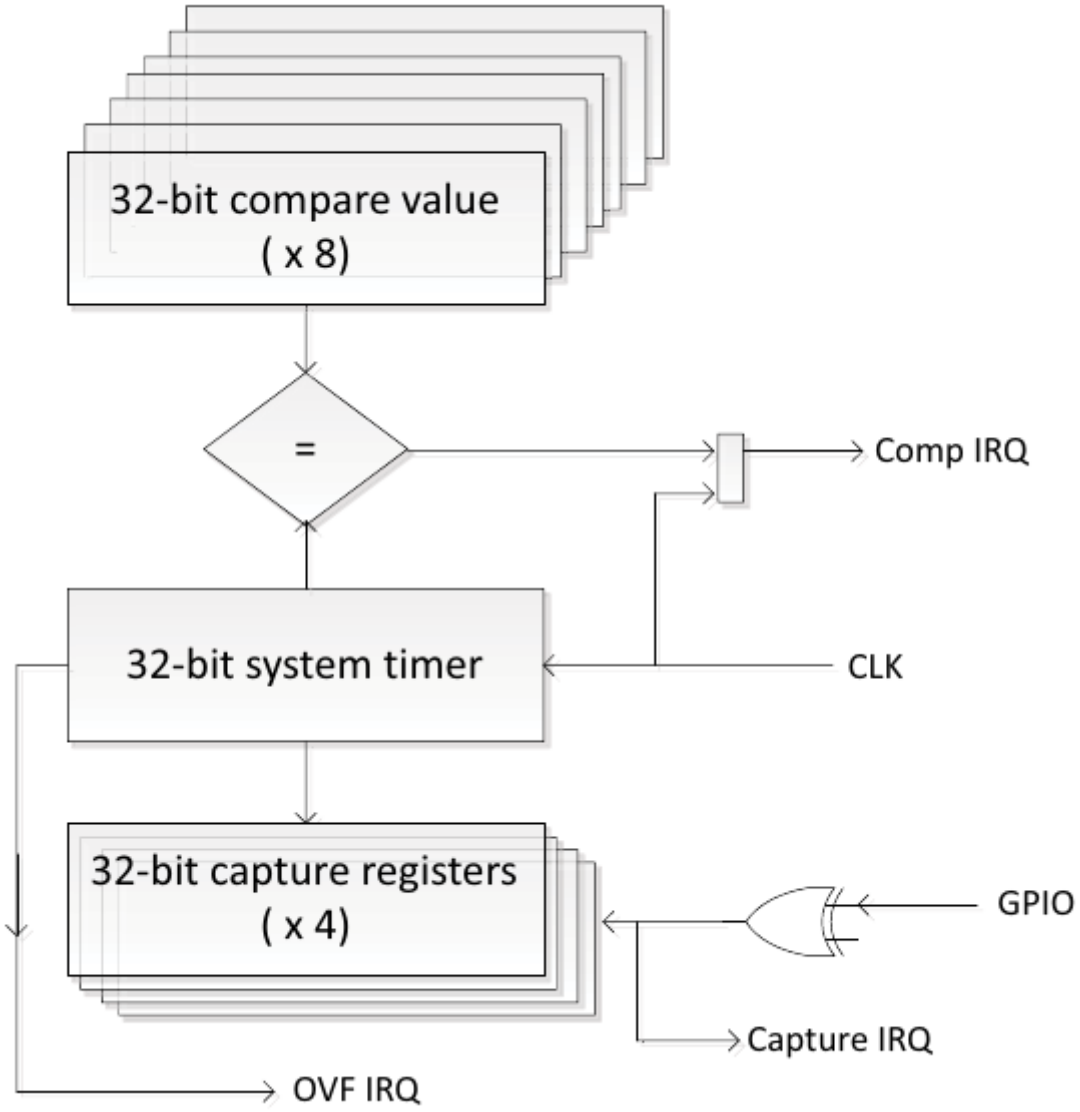

Figure 7.2: Block Diagram of the Apollo 3 System Timer

The Apollo 3 SOC supplies a system timer peripheral which consists of a register which counts at a programmable frequency and has eight compare registers which indicate when there is a match between the counter and the compare register. This peripheral was discussed in Chapter 2 where it was initialized.

A system timer compare register is the hardware basis for operating a timer queue instance. There are eight timer compare registers[1]. The operating rules for the compare registers are slightly unusual.

[1] There are also capture registers in the system timer block, but they are not used here

- The value **written** to the compare registers is the offset, in timer ticks, from *now* to when the comparitor is to match. The addition of the offset to the current value of the timer counter, yielding the compare register match value, is done in hardware. This is quite a nice feature and essential given the high frequency the counter clock can support.
- The values **read** from the compare register is the counter value at which the compare register will match, *i.e.* after writing an offset, you read by the sum computed in hardware.
- Each compare register is wired to a separate NVIC interrupt line. This is somewhat unusual as most SOC designs wire peripherals to only a single NVIC interrupt line and then use an internal register to indicate the source of the interrupt. This arrangement gives software a separate IRQ vector directly connected to the compare register interrupt. This is also a nice feature for faster response to compare register matches.

Two compare registers of the system timer peripheral are allocated to run two instances of timer queues. For one timer queue instance, there is a definite plan for its use which is considered in Part II of the book. The other instance is available for any application purpose which needs to perform time ordered sequencing of actions.

Given the rules of operation for the compare registers, the necessary hardware controls are few.

```
<<dev realm timq proxy: data type declarations>>=
typedef struct {
    IRQn_Type irq_num ;
    uint32_t volatile *cmp_reg ;
    uint32_t intr_mask ;
    uint32_t cmp_enable ;
} TIMQ_HdwrControls ;
```

irq_num
: The interrupt number associated with the system timer compare register.

cmp_reg
: A pointer to the memory mapped I/O for the compare register.

intr_mask
: A bit mask with a single set bit which corresponds to the interrupt control registers for the system timer. The interrupt status, set, and clear registers all have the same bit-field arrangement.

cmp_enable
: A bit mask with a single set bit which corresponds to the enable control for the compare register used for the timer queue.

All the other hardware controls are singleton registers within the system timer peripheral and are accessed directly as necessary.

Timer Queue Control Block

There are four aspects of the timer queue design which must be managed:

1. The linked list of queued timer elements.
2. Allocation and deallocation of queue elements as they are inserted and removed from the timer queue itself.
3. Sending background notifications when the status of a timer element changes.
4. Programming the actions of a compare register in the system timer peripheral.

This leads to the following members for a data structure used to control the timer queue operations.

```
<<dev realm timq proxy: data type declarations>>=
typedef struct {
    CLL_Links queue ;
    BKAL_ControlBlock *element_allocator ;
    SVC_DevTimqNotification notification ;
    TIMQ_HdwrControls hdwr_controls ;
} TIMQ_ControlBlock ;
```

queue
The head of a circularly linked list of timer queue elements.

element_allocator
A pointer to the allocation control block for obtaining queue elements to place in the timer queue.

notification
The notification data sent in the background notification which is generated upon the change of state in a queue element.

hdwr_controls
The set of registers and constants needed to program a compare register in the Apollo 3 system timer peripheral.

The values of the control blocks are initialized at compile time to save ourselves yet another initialization function. The first two compare registers in the system timer peripheral are allocated to supply the timing for the queues.

```
<<dev realm timq proxy: static data definitions>>=
static TIMQ_ControlBlock timer_queues[TIMQ_INSTANCES] = {
    [0] = {
        .queue = {
            .next = &timer_queues[0].queue,                    // ❶
            .prev = &timer_queues[0].queue,
        },
        .element_allocator = &inst0_allocator,
        .notification = {
            .block_size = sizeof(SVC_DevTimqNotification),
            .notify_proxy = NULL,
            .notify_closure = 0,
            .device_class = DEV_TIMQ_CLASS,
            .device_instance = 0,
            .element_id = -1,
            .status = timq_discarded,
        },
        .hdwr_controls = {
            .irq_num = STIMER_CMPR0_IRQn,                      // ❷
            .cmp_reg = &CTIMER->SCMPR[0],
            .intr_mask = CTIMER_STMINTEN_COMPAREA_Msk,
            .cmp_enable = CTIMER_STCFG_COMPARE_A_EN_Msk,
        },
    },
    [1] = {
        .queue = {
            .next = &timer_queues[1].queue,
            .prev = &timer_queues[1].queue,
        },
        .element_allocator = &inst1_allocator,
        .notification = {
            .block_size = sizeof(SVC_DevTimqNotification),
            .notify_proxy = NULL,
            .notify_closure = 0,
            .device_class = DEV_TIMQ_CLASS,
            .device_instance = 1,
```

```
            .element_id = -1,
            .status = timq_discarded,
        },
        .hdwr_controls = {
            .irq_num = STIMER_CMPR1_IRQn,
            .cmp_reg = &CTIMER->SCMPR[1],
            .intr_mask = CTIMER_STMINTEN_COMPAREB_Msk,
            .cmp_enable = CTIMER_STCFG_COMPARE_B_EN_Msk,
        },
    },
} ;
```

❶ Initialize the queue to be empty by pointing back to itself.

❷ The hardware pre-processor symbols come directly from the CMSIS header file.

Timer Queue Operators

This section defines the set of functions which act as operators on the timer queue control blocks.

Note
All the timer queue operations assume that the IRQ Handler for the corresponding timer compare register does not execute during the operation. This happens either because the IRQ handler itself is currently executing (recall all IRQ handlers run at the same priority in this design) or that the foreground proxy functions (associated with timer queue requests coming from the background via the SVC exception) has explicitly disabled the interrupt. The operations share data with the IRQ handler and must implement a critical section with respect to the associated interrupt. This is made visibly distinct in the code below.

A timer queue operates in one of two states.

1. The queue is running. The hardware timing peripheral is counting down the expiration time of the element at the head of the queue. In this case, the `expire_time` member of the element at the head of the timer queue is zero as its expiration time was moved into the hardware. The head element requires no additional expiration time past that loaded into the system timer compare register.
2. The queue is stopped, implying that the timer compare register is not being used. For this case, any residual time remaining in the timer register is placed back into the `expire_time` member of the element at the head of the queue. Stopping the timer queue should not be confused with stopping the timer peripheral. The timer peripheral counter is never stopped.

The timer queue must be stopped for any operations on the timer queue except for expiring queue elements, This is required in order to correctly insert or remove an element keeping the ordering relationship among the elements correct. For example, when inserting a new element, the requested expiration time might be shorter than the residual time of the element at the head of the queue.

Since starting and stopping the timer queue involve interactions with the system timer peripheral, the coding requires more than the usual care. The hardware counters run asynchronously to the processor. This means that there is a race condition between what the timer is doing compared to what the processor sees. The difference arises because the processor must execute instructions to view the actions of the peripheral through its memory mapped registers. This can be obscured, especially in the case of “C” language statements, since there is not necessarily a one-to-one correspondence from language statements to machine instructions which are accessing the timer peripheral.

```
<<dev realm timq proxy: forward references>>=
static void
timq_stop(
    TIMQ_ControlBlock *const tcb) ;
```

tcb
A pointer to the timer queue control block to be stopped.

The `timq_stop` function halts timer queue operations for the queue given by `tcb` and places the residual unexpired time of the element at the head of the queue into that element's `expire_time` member.

To stop the timer queue, the residual time remaining in the hardware must be determined. The residual time is the difference between the compare register value and the timer counter. After reading and computing that difference, a check of the interrupt status is made to determine if the compare register matched during the time it took to obtain the residual time. If the interrupt status is set (*N.B.* since the interrupt is disabled at this point, execution has not been preempted), then the race was "lost" and simply declare the queue element as expired. The expiration is indicated by a `expire_time` element value of 0.

```
<<dev realm timq proxy: static function definitions>>=
static void
timq_stop(
    TIMQ_ControlBlock *const tcb)
{
    assert(tcb != NULL) ;

    if (!cll_empty(&tcb->queue)) {
        uint32_t counter = CTIMER->STTMR ;
        uint32_t expire = *tcb->hdwr_controls.cmp_reg ;

        TIMQ_Element *head = CONTAINER_OF(cll_begin(&tcb->queue),
                TIMQ_Element, links) ;
        head->expire_time = expire - counter ;                    // ❶

        bool expired = btwd_test_reg_field(&CTIMER->STMINTSTAT,
                tcb->hdwr_controls.intr_mask) ;
        if (expired) {
            head->expire_time = 0 ;                               // ❷
        }
    }
}
```

❶ Because of the two's complement representation, this expression works even if the expiration time wrapped around when it was loaded into the compare register.

❷ If the interrupt status indicates a match, then there is no residual time.

```
<<dev realm timq proxy: forward references>>=
static void
timq_start(
    TIMQ_ControlBlock *const tcb) ;
```

tcb
A pointer to the timer queue control block to be started.

The `timq_start` function starts the timer queue given by `tcb` by placing the `expire_time` value of the element at the head of the timer queue into the hardware peripheral which is used for the timer queue.

Starting the timer queue is easier. The hardware interactions must be ordered such that the compare register interrupt is cleared and ready to react before loading the compare register. This insures that the interrupt can be latched before installing the time value into the compare registers. For small expire times, this is important. After loading the compare register, the interrupt is enabled.

```
<<dev realm timq proxy: static function definitions>>=
static void
timq_start(
    TIMQ_ControlBlock *const tcb)
{
    assert(tcb != NULL) ;

    timq_expire_elements(tcb) ;                                          // ❶

    if (!cll_empty(&tcb->queue)) {
        TIMQ_Element *head = CONTAINER_OF(cll_begin(&tcb->queue),
                TIMQ_Element, links) ;
        assert(head->expire_time != 0) ;

        timq_set_expiration_time(&tcb->hdwr_controls, head->expire_time) ;      // ❷
        head->expire_time = 0 ;
    } else {
        btwd_clear_reg_field(&CTIMER->STCFG, tcb->hdwr_controls.cmp_enable) ;
        btwd_clear_reg_field(&CTIMER->STMINTEN, tcb->hdwr_controls.intr_mask) ; // ❸
    }
}
```

❶ Before starting the timer queue, any elements which have an `expire_time` of zero must be expired. Expired elements could come either when the timing was stopped (see `timq_stop` above) or if a timer element with a zero expire time was inserted or updated. In either case, the timer queue must be placed into a state where it has a non-zero value as the `expire_time` for the element at the head of the queue.

❷ Setting the time value into the compare register was factored into a separate function because of a chip erratum. See below.

❸ Note that the compare register interrupt is left disabled when the timer queue is empty.

It is common for SOC's to have errors in their circuitry. These so called, "errata", are typically cataloged by the chip manufacturer along with any available workarounds. In this case the Ambique Apollo 3 erratum **ERR014** [2] details the problem and a workaround. The workaround is in the HAL code of the Ambique SDK and the basic processing from the HAL code (which follows the erratum prescriptions) has been used in the following function.

```
<<dev realm timq proxy: forward references>>=
static void
timq_set_expiration_time(
    TIMQ_HdwrControls const *const hdctrl,
    TIMQ_TimeTicks expire_time) ;

<<dev realm timq proxy: static function definitions>>=
static void
timq_set_expiration_time(
    TIMQ_HdwrControls const *const hdctrl,
    TIMQ_TimeTicks expire_time)
{
    bool done = false ;
    unsigned tries = 0 ;
#   define STTMR_ACCESS_LATENCY 10                                  /* ❶ */

    uint32_t intr_mask = hdctrl->intr_mask ;
```

[2] *Apollo3 Blue MCU Errata List*, Doc.ID:SE-A3_2p09, Revision 2.0, July 2019

```
    uint32_t mask = stwd_begin_critical_section() ;                 // ❷
// BEGIN CRITICAL SECTION

    do {
        uint32_t expected_lower = CTIMER->STTMR + expire_time ;
        uint32_t expected_upper = expected_lower + STTMR_ACCESS_LATENCY ;
        CTIMER->STMINTCLR = intr_mask ;
        btwd_set_reg_field(&CTIMER->STMINTEN, intr_mask) ;
        btwd_set_reg_field(&CTIMER->STCFG, hdctrl->cmp_enable) ;
        *hdctrl->cmp_reg = expire_time ;
        uint32_t compare = *hdctrl->cmp_reg ;
        if (compare >= expected_lower && compare < expected_upper) {
            done = true ;
        } else {
            btwd_clear_reg_field(&CTIMER->STCFG, hdctrl->cmp_enable) ;
            btwd_clear_reg_field(&CTIMER->STMINTEN, intr_mask) ;
            tries += 1 ;
        }
    } while (!done && tries < 4) ;

    if (!done) {
        panic("unable to set system timer expiration time") ;
    }

    stwd_end_critical_section(mask) ;
// END CRITICAL SECTION

#   undef STTMR_ACCESS_LATENCY
}
```

❶ 10 ticks is what is used in the Amique HAL. No explanation is given for why that particular value was chosen. For the clock frequency at which the system timer is run, 10 ticks is greater than 300 μs. This seems excessive.

❷ *N.B.* this uses a critical section with all interrupts disabled. Since the workaround for the race condition in the hardware requires repeated iterations in software, the workaround code must execute to completion without interruption.

Where the start and stop operations interacted with the hardware, the insert, remove, and remaining operations are only concerned about searching the timer queue and maintaining the relative expiration times in proper order. Insertion means walking the queue, subtracting the expiration times from the requested time until the place in the queue is found where an element with the requested time belongs.

```
<<dev realm timq proxy: forward references>>=
static void
timq_insert_element(
    TIMQ_ControlBlock *const tcb,
    TIMQ_Element *element) ;
```

tcb
A pointer to the timer queue control block to which the insertion is made.

element
A pointer to a timer queue element which is inserted into `tcb`.

The `timq_insert_element` function inserts the timer queue element given by `element` into the timer queue pointed to by `tcb`. It is assumed that the `expire_time` and `reload_time` members of the `element` structure have been set to the requested expiration and reload times, respectively.

```
<<dev realm timq proxy: static function definitions>>=
static void
timq_insert_element(
    TIMQ_ControlBlock *const tcb,
    TIMQ_Element *element)
{
    assert(tcb != NULL) ;
    assert(element != NULL) ;

    CLL_Links *const end = cll_end(&tcb->queue) ;
    CLL_Links *iter = cll_begin(&tcb->queue) ;
    for ( ; iter != end ; iter = iter->next) {
        TIMQ_Element *current = CONTAINER_OF(iter, TIMQ_Element, links) ;
        if (element->expire_time < current->expire_time) {          // ❶
            current->expire_time -= element->expire_time ;          // ❷
            break ;
        } else {
            element->expire_time -= current->expire_time ;
        }
    }

    cll_insert(&element->links, iter) ;                             // ❸
}
```

❶ By making the comparison strictly less than, the ordering of the expiration of the elements that, coincidently, are meant to expire at the same time is maintained to be the same as the order of insertion.

❷ This is the right spot, but the expiration time of the new entry must be subtracted from those which follow in order not to add time to the trailing entries.

❸ The linked list insert causes the element to be inserted *before* the list location found by the search.

Note particularly what happens in the above code if the inserted element's `expire_time` is zero, *i.e.* if `element->expire_time` is zero. It is inserted into the queue as expected, but behind any other elements that might have already expired. This maintains an ordering where elements expired by virtue of the timer precede those expired by virtue of supplied arguments. This is the desired order. The implication is that requesting an expiration time of zero expires the element immediately, but in the correct order if there are other elements that have expired, coincidentally, to the request.

Removing a queue element is similar. In the removal case, any expiration time of the removed element must be added to the next element in the list. Of course, the boundary condition of removing the last element in the queue requires a conditional test. Note that removing an element from the queue does *not* automatically return the element to the storage pool. An update request uses this function to remove the queue element, but then inserts the queue element with a new expiration time.

```
<<dev realm timq proxy: forward references>>=
static void
timq_remove_element(
    TIMQ_ControlBlock *const tcb,
    TIMQ_Element *element) ;
```

tcb
A pointer to the timer queue control block from which the removal is made.

element
A pointer to a timer queue element which is removed.

The `timq_remove_element` removes the queue element pointed to by `element` from the timer queue referenced by `tcb`.

```
<<dev realm timq proxy: static function definitions>>=
static void
timq_remove_element(
    TIMQ_ControlBlock *const tcb,
    TIMQ_Element *element)
{
    assert(tcb != NULL) ;
    assert(element != NULL) ;

    CLL_Links *const end = cll_end(&tcb->queue) ;
    for (CLL_Links *iter = cll_begin(&tcb->queue) ;
            iter != end ; iter = iter->next) {
        TIMQ_Element *current = CONTAINER_OF(iter, TIMQ_Element, links) ;
        if (current == element) {
            if (iter->next != end) {                                    // ❶
                TIMQ_Element *next_element =
                        CONTAINER_OF(iter->next, TIMQ_Element, links) ;
                next_element->expire_time += current->expire_time ;
            }
            cll_remove(&element->links) ;
            break ;
        }
    }
}
```

❶ If the removed element is not the last one in the queue, any expiration time associated with it must be added to its next neighbor.

Note that if it were *not* required to handle any residual time in the removed element, the iteration across the timer queue would be unnecessary since an element of a doubly-linked list can be removed by direct pointer manipulation.

Computing the remaining time for an element is similar to the inverse of what happens at insertion time. Where insertion subtracted off the preceding expiration times, here the queue is walked adding together all the expiration times until the matching element is found.

```
<<dev realm timq proxy: forward references>>=
static TIMQ_TimeTicks
timq_remaining_time(
    TIMQ_ControlBlock *const tcb,
    TIMQ_Element *element) ;
```

tcb
A pointer to the timer queue control block to which `element` is queued.

element
A pointer to a timer queue element for which the remaining time before expiration is computed.

The `timq_remaining_time` function returns the minimum number of timer ticks which must elapse before `element` expires.

```
<<dev realm timq proxy: static function definitions>>=
static TIMQ_TimeTicks
timq_remaining_time(
    TIMQ_ControlBlock *const tcb,
    TIMQ_Element *element)
{
    assert(tcb != NULL) ;
```

```
    assert(element != NULL) ;

    TIMQ_TimeTicks remaining_time = 0 ;

    CLL_Links *const end = cll_end(&tcb->queue) ;
    for (CLL_Links *iter = cll_begin(&tcb->queue) ;
            iter != end ; iter = iter->next) {
        TIMQ_Element *current = CONTAINER_OF(iter, TIMQ_Element, links) ;
        remaining_time += current->expire_time ;
        if (current == element) {
            break ;
        }
    }

    return remaining_time ;
}
```

There are two operations associated with sending background notifications. When an element expires, the queue is traversed finding all the elements with zero expiration times and a background notification is posted. Once a notification has been sent for an element, it may be returned to the storage pool if it is a one-shot expiration or re-inserted into the timer queue if it has periodic expiration.

```
<<dev realm timq proxy: forward references>>=
static void
timq_expire_elements(
    TIMQ_ControlBlock *const tcb) ;
```

tcb
A pointer to the timer queue control block which contains the elements to expire.

The `timq_expire_elements` function sends background notifications to all timer queue elements which have expired.

```
<<dev realm timq proxy: static function definitions>>=
static void
timq_expire_elements(
    TIMQ_ControlBlock *const tcb)
{
    assert(tcb != NULL) ;

    CLL_Links *const end = cll_end(&tcb->queue) ;
    for (CLL_Links *iter = cll_begin(&tcb->queue) ; iter != end ; ) {
        TIMQ_Element *current = CONTAINER_OF(iter, TIMQ_Element, links) ;
        iter = iter->next ;                                                    // ❶

        if (current->expire_time == 0) {
            cll_remove(&current->links) ;

            tcb->notification.element_id =
                    blk_ref_to_index(tcb->element_allocator, current) ;        // ❷
            tcb->notification.notify_closure = current->notify_closure ;
            if (current->reload_time == 0) {
                blk_free(tcb->element_allocator, current) ;
                tcb->notification.status = timq_expired ;
            } else {
                current->expire_time = current->reload_time ;
                timq_insert_element(tcb, current) ;
                tcb->notification.status = timq_reloaded ;
            }
```

```
            bool sent = send_bg_notification(&tcb->notification) ;
            if (!sent) {
                panic("failed to post timer queue background notification: "
                        "inst = %u, element = %d",
                        tcb->notification.device_instance,
                        tcb->notification.element_id) ;
            }

        } else {
            break ;                                                    // ❸
        }
    }
}
```

❶ Since an element may be removed during the queue traversal and element removal invalidates the list iterator, the iterator must be advanced_before_ removing any element.

❷ The block allocator can calculate the array index of any of its allocated blocks.

❸ The first non-zero expiration time marks the end of those elements which require notifications.

If the timer queue is “closed” and there are still elements in the queue, a background notification is posted for all elements. The notification has a status indicating that the queue element has been discarded. This informs background processing that no notifications for those elements will ever come.

```
<<dev realm timq proxy: forward references>>=
static void
timq_close_elements(
    TIMQ_ControlBlock *const tcb) ;
```

tcb
A pointer to the timer queue control block which contains elements to discard.

The `timq_close_elements` function sends background notifications to all elements in the timer queue indicating the status of the element as `discarded`.

```
<<dev realm timq proxy: static function definitions>>=
static void
timq_close_elements(
    TIMQ_ControlBlock *const tcb)
{
    tcb->notification.status = timq_discarded ;

    CLL_Links *const end = cll_end(&tcb->queue) ;
    for (CLL_Links *iter = cll_begin(&tcb->queue) ; iter != end ; ) {
        TIMQ_Element *current = CONTAINER_OF(iter, TIMQ_Element, links) ;

        iter = iter->next ;                                            // ❶
        cll_remove(&current->links) ;

        tcb->notification.element_id =
                blk_ref_to_index(tcb->element_allocator, current) ;
        tcb->notification.notify_closure = current->notify_closure ;
        bool sent = send_bg_notification(&tcb->notification) ;
        if (!sent) {
            panic("failed to post timer queue background notification: "
                    "instance = %u, element = %d",
```

```
                    tcb->notification.device_instance,
                    tcb->notification.element_id) ;
        }

        blk_free(tcb->element_allocator, current) ;
    }
}
```

❶ Again, linked list removal invalidates the iterator and it must be advanced *before* removal.

Background Timing Queue Requests

All the foundation is in place to run the timer queue. What remains is to provide an interface to background processing to make timer queue requests. This interface follows the pattern for device control already seen.

The following figure shows the interactions between the device realm foreground proxy functions and the peripheral hardware.

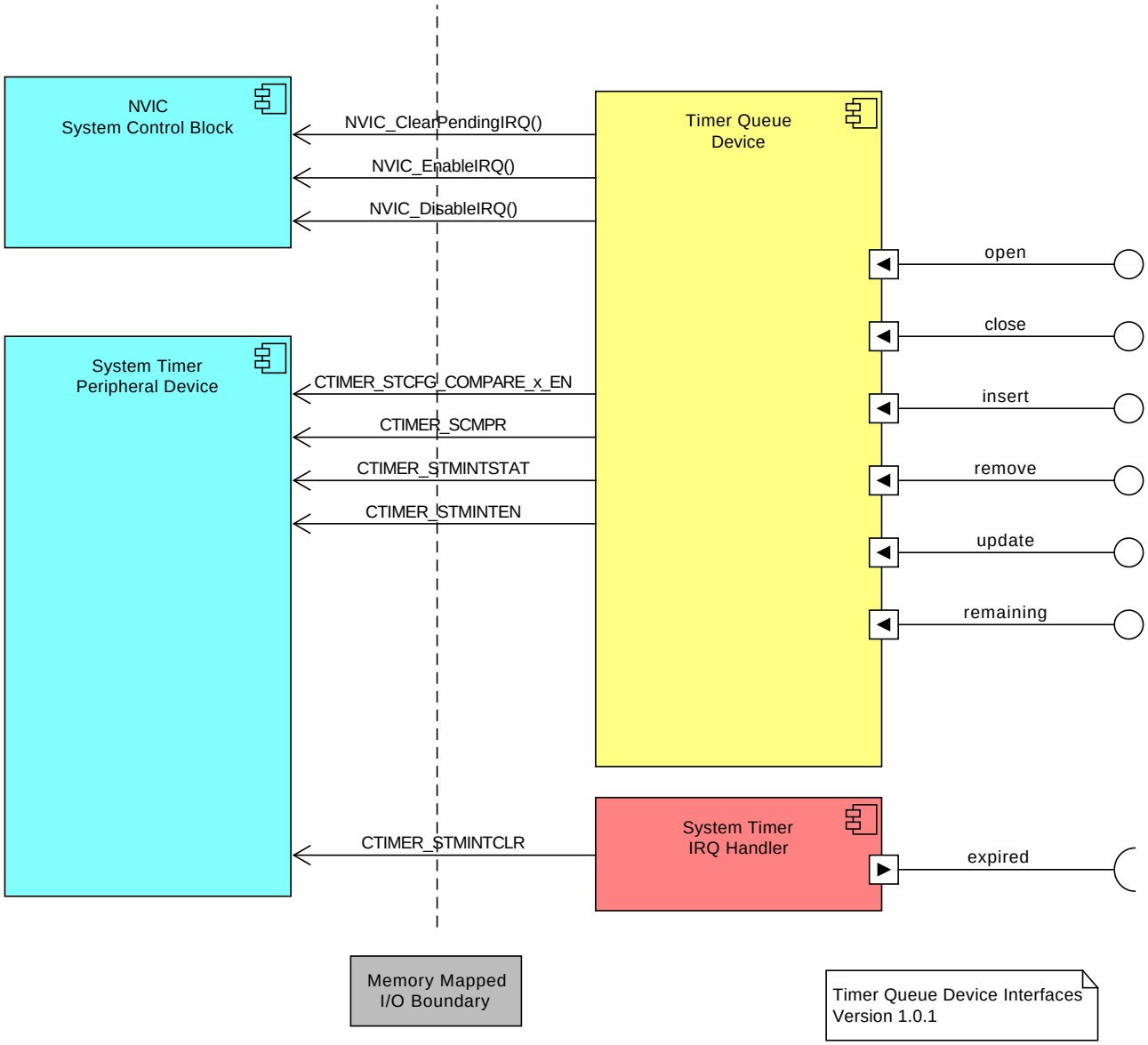

Figure 7.3: Timer Queue Interface Components

Timer queue request encoding

First, the timer queue request operations are encoded as small integer numbers. Since the enumerator values are used for dispatching the corresponding function, they start at zero and the values are sequential.

```
<<dev realm timq param: data type declarations>>=
enum SVC_timqRequest {
    timq_open = 0,
    timq_close,
    timq_insert,
    timq_remove,
    timq_update,
    timq_remaining,
```

```
    timq_operation_count        // last
} ;
```

Open a Timer Queue

```
<<dev svc timq req: external declarations>>=
extern int
dev_timq_open(
    SVC_DevInstance timq,
    SVC_DevTimqNotifyProxy proxy) ;
```

timq
The instance number which identifies which timer queue is to be opened.

proxy
A pointer to a notification proxy function which is to be invoked during background processing when a timer queue entry changes status.

The `dev_timq_open` function makes the timer queue given by, `timq`, ready for timing queue operations. When a timing queue element expires, the background notification contains the proxy function, `proxy`, which is to be invoked to process the element expiration in the background. This function must be invoked before any other operations on the timer queue.

The input parameters must be transferred across the `SVC` interface and need an appropriate data structure to do so.

```
<<dev realm timq param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevTimqOpenInput,
    SVC_DevTimqNotifyProxy proxy ;
) ;
```

The background function which marshals the request follows the same pattern used previously — fill in the parameter structures and make a device realm `SVC` call.

```
<<dev svc timq req: external definitions>>=
int
dev_timq_open(
    SVC_DevInstance timq,
    SVC_DevTimqNotifyProxy proxy)
{
    SVC_DevTimqOpenInput input = {
        .block_size = sizeof(SVC_DevTimqOpenInput),
        .proxy = proxy,
    } ;

    SVC_DevRequest timq_req = dev_req_encode(DEV_TIMQ_CLASS, timq_open, timq) ;
    return dev_realm_svc_call(timq_req, &input, NULL, NULL) ;
}
```

The following function is the foreground proxy for `dev_timq_open`.

```
<<dev realm timq proxy: static function definitions>>=
static int
dev_realm_timq_open(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
```

```
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    TIMQ_ControlBlock *const tcb = &timer_queues[inst] ;
    rtcheck_NULL_return(tcb->notification.notify_proxy, -ERR_OPERATION_FAILED) ;
    rtcheck_return(cll_empty(&tcb->queue), -ERR_OPERATION_FAILED) ;

    SVC_DevTimqOpenInput tq_input ;
    int result = copy_in_svc_param(input, sizeof(tq_input), &tq_input) ;

    rtcheck_zero_return(result, result) ;
    rtcheck_not_NULL_return(tq_input.proxy, -ERR_INVALID_PARAM) ;

    cmp_reg_disable_irq(tcb) ;

    tcb->notification.notify_proxy = (SVC_DevNotifyProxy)tq_input.proxy ;
    tcb->notification.notify_closure = 0 ;
    tcb->notification.element_id = -1 ;
    tcb->notification.status = timq_discarded ;

    return 0 ;
}
```

Basic timer peripheral compare register operations are placed in a function.

```
<<dev realm timq proxy: static inline functions>>=
static inline void
cmp_reg_disable_irq(
    TIMQ_ControlBlock *const tcb)
{
    NVIC_DisableIRQ(tcb->hdwr_controls.irq_num) ;
    btwd_clear_reg_field(&CTIMER->STCFG, tcb->hdwr_controls.cmp_enable) ;
    btwd_clear_reg_field(&CTIMER->STMINTEN, tcb->hdwr_controls.intr_mask) ;
    NVIC_ClearPendingIRQ(tcb->hdwr_controls.irq_num) ;
}
```

Close a Timer Queue

```
<<dev svc timq req: external declarations>>=
extern int
dev_timq_close(
    SVC_DevInstance timq) ;
```

timq
The instance number which identifies which timer queue is to be close.

The `dev_timq_close` function decommissions the timer queue instance given by `timq`. After invoking `dev_timq_close` no further timer expiration notifications are made. Any queued timer elements are notified with the status field in the notification set to `timq_discarded`.

The background request function is trivial since there are no arguments to the request.

```
<<dev svc timq req: external definitions>>=
int
dev_timq_close(
```

```
    SVC_DevInstance timq)
{
    SVC_DevRequest timq_req = dev_req_encode(DEV_TIMQ_CLASS, timq_close, timq) ;
    return dev_realm_svc_call(timq_req, NULL, NULL, NULL) ;
}
```

The foreground proxy function returns any lingering queue elements and restores the state of the timer control block.

```
<<dev realm timq proxy: static function definitions>>=
static int
dev_realm_timq_close(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(input == NULL) ; (void)input ;
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    TIMQ_ControlBlock *const tcb = &timer_queues[inst] ;

    cmp_reg_disable_irq(tcb) ;

    timq_close_elements(tcb) ;
    assert(cll_empty(&tcb->queue)) ;

    tcb->notification.notify_proxy = NULL ;
    tcb->notification.notify_closure = 0 ;
    tcb->notification.element_id = -1 ;

    return 0 ;
}
```

Insert a Timer Element

```
<<dev svc timq req: external declarations>>=
extern int
dev_timq_insert(
    SVC_DevInstance timq,
    TIMQ_TimeTicks expire_time,
    TIMQ_TimeTicks reload_time,
    SVC_DevNotifyClosure notify_closure) ;
```

timq
The instance number which identifies which timer queue into which a new timing element is inserted.

expire_time
The minimum number of timer ticks which must elapse before an expiration notification is sent. An `expire_time` value of 0, causes the newly inserted element to expire immediately.

reload_time
The minimum number of timer ticks which must elapse before an expiration notification is sent on the second and subsequent periods. A `reload_time` of zero indicates that the timer element issues only a single notification. A non-zero `reload_time` value indicates the inserted element is to issue repeated notifications every `reload_time` ticks after the element expires for the first time.

notify_closure
A value returned in any background notification associated with the newly inserted timer element.

The `dev_timq_insert` function inserts a new element into the timer queue given by, `timq`. The element is set to expire at, `expire_time`, timer ticks from now. If the timer element is intended to be periodic, then `reload_time` is supplied as non-zero and the second and subsequent expirations of the element happen `reload_time` ticks after the first expiration. Note, `expire_time` and `reload_time` need not be the same value for a periodic timer queue element. It is possible to use `expire_time` to synchronize to some ongoing period which has partially elapsed and subsequent notifications are issued every `reload_time` ticks.

A negative return value of `dev_timq_insert` indicates an error. A non-negative return value gives an identifier for the timer queue element which is used as an argument in other timer queue operation functions.

The scaling factor for timer ticks depends upon the frequency of the system timer peripheral clock. The clock ticks at 32786 Hz and is held in a 32 bit register. The queue element times are treated as an unsigned fixed binary point UQ17.15 number in units of seconds. Scaling functions are provided below.

```
<<dev realm timq param: data type declarations>>=
typedef uint32_t TIMQ_TimeTicks ;
```

The input parameter for the insert function requires two tick times.

```
<<dev realm timq param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevTimqInsertInput,
    TIMQ_TimeTicks expire_time ;
    TIMQ_TimeTicks reload_time ;
    SVC_DevNotifyClosure notify_closure ;
) ;
```

The output parameter returns the queue element identifier.

```
<<dev realm timq param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevTimqInsertOutput,
    TIMQ_ElementID element_id ;
) ;
```

The background request function follows the familiar pattern.

```
<<dev svc timq req: external definitions>>=
```

```
int
dev_timq_insert(
    SVC_DevInstance timq,
    TIMQ_TimeTicks expire_time,
    TIMQ_TimeTicks reload_time,
    SVC_DevNotifyClosure notify_closure)
{
    SVC_DevTimqInsertInput input = {
        .block_size = sizeof(SVC_DevTimqInsertInput),
        .expire_time = expire_time,
        .reload_time = reload_time,
        .notify_closure = notify_closure,
    } ;
    SVC_DevTimqInsertOutput output = {
        .block_size = sizeof(SVC_DevTimqInsertOutput),
        .element_id = -1,
    } ;

    SVC_DevRequest timq_req = dev_req_encode(DEV_TIMQ_CLASS, timq_insert, timq) ;
    int status = dev_realm_svc_call(timq_req, &input, &output, NULL) ;

    return status < 0 ? status : output.element_id ;
}
```

The device realm foreground proxy function is a wrapper around the `timq_insert_element` operation. It is necessary to allocate a timer queue element and fill in the parameters. With an element in hand, the timer queue operation sequence is stop, insert, start. As shown below and in subsequent proxy functions, the core of the logic of the queue element proxy functions is contained in the timer queue operations defined previously.

```
<<dev realm timq proxy: static function definitions>>=
static int
dev_realm_timq_insert(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    TIMQ_ControlBlock *const tcb = &timer_queues[inst] ;
    rtcheck_not_NULL_return(tcb->notification.notify_proxy, -ERR_OPERATION_FAILED) ;

    SVC_DevTimqInsertInput tq_input ;
    int result = copy_in_svc_param(input, sizeof(tq_input), &tq_input) ;
    rtcheck_zero_return(result, result) ;

    IRQn_Type irq_num = tcb->hdwr_controls.irq_num ;
    (void)stwd_begin_interrupt_section(irq_num) ;
//BEGIN INTERRUPT SECTION

    TIMQ_Element *element = blk_allocate(tcb->element_allocator) ;
    assert(element != NULL) ;
    if (element != NULL) {
        element->expire_time = tq_input.expire_time ;
        element->reload_time = tq_input.reload_time ;
        element->notify_closure = tq_input.notify_closure ;

        timq_stop(tcb) ;
        timq_insert_element(tcb, element) ;
        timq_start(tcb) ;
```

```
        SVC_DevTimqInsertOutput tq_output = {
            .block_size = sizeof(tq_output),
            .element_id = blk_ref_to_index(tcb->element_allocator, element),
        } ;
        result = copy_out_svc_result(output, sizeof(tq_output), &tq_output) ;
    } else {
        result = -ERR_OPERATION_FAILED ;
    }

    stwd_end_interrupt_section(irq_num, true) ;
//END INTERRUPT SECTION

    return result ;
}
```

Remove a Timer Element

```
<<dev svc timq req: external declarations>>=
extern int
dev_timq_remove(
    SVC_DevInstance timq,
    TIMQ_ElementID element_id) ;
```

timq
The instance number which identifies which timer queue from which a timing element is removed.

element_id
An identifier of a timer queue element. This value must be the same as some return value from a successful invocation of `dev_timq_insert`.

The `dev_timq_remove` function removes a timer queue element given by `element_id` from the timer queue given by `timq`.

A return value of zero indicates that `element_id` was successfully removed. A negative return values indicates an error occurred.

The input to the remove request requires passing the element ID for the element to be removed.

```
<<dev realm timq param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevTimqRemoveInput,
    TIMQ_ElementID element_id ;
) ;
```

```
<<dev svc timq req: external definitions>>=
int
dev_timq_remove(
    SVC_DevInstance timq,
    TIMQ_ElementID element_id)
{
    SVC_DevTimqRemoveInput input = {
        .block_size = sizeof(SVC_DevTimqRemoveInput),
        .element_id = element_id,
    } ;

    SVC_DevRequest timq_req = dev_req_encode(DEV_TIMQ_CLASS, timq_remove, timq) ;
    return dev_realm_svc_call(timq_req, &input, NULL, NULL) ;
}
```

Similar to the insert operation, the device realm foreground proxy function is a wrapper around the `timeq_remove_element` operation. Here the sequence of timer queue operations is stop, remove, start.

```
<<dev realm timq proxy: static function definitions>>=
static int
dev_realm_timq_remove(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    TIMQ_ControlBlock *const tcb = &timer_queues[inst] ;

    SVC_DevTimqRemoveInput tq_input ;
    int result = copy_in_svc_param(input, sizeof(tq_input), &tq_input) ;
    rtcheck_zero_return(result, result) ;

    IRQn_Type irq_num = tcb->hdwr_controls.irq_num ;
    (void)stwd_begin_interrupt_section(irq_num) ;
//BEGIN INTERRUPT SECTION

    TIMQ_Element *element = blk_index_to_ref(tcb->element_allocator,
            tq_input.element_id) ;
    if (element != NULL) {
        timq_stop(tcb) ;
        timq_remove_element(tcb, element) ;
        timq_start(tcb) ;

        blk_free(tcb->element_allocator, element) ;
        result = 0 ;
    } else {
        result = -ERR_OPERATION_FAILED ;
    }

    stwd_end_interrupt_section(irq_num, true) ;
//END INTERRUPT SECTION

    return result ;
}
```

Update a Timer Element

```
<<dev svc timq req: external declarations>>=
extern int
dev_timq_update(
    SVC_DevInstance timq,
    TIMQ_ElementID element_id,
    TIMQ_TimeTicks expire_time,
    TIMQ_TimeTicks reload_time) ;
```

timq
The instance number which identifies which timer queue into which a new timing element is inserted.

element_id
An identifier of a timer queue element. This value must be the same as some return value from a successful invocation of `dev_timq_insert`.

expire_time
The minimum number of timer ticks which must elapse before an expiration notification is sent.

reload_time
The minimum number of timer ticks which must elapse on the second and subsequent periods before an expiration notification is sent. A `reload_time` of zero indicates that the timer element issues only a single notification.

The `dev_timq_update` function modifies, in place, the timing parameters of the timer queue element given by, `element_id` which resides in the timer queue given by `timq`. This function performs the logical equivalent of removing the element from the timer queue and re-inserting the same element with new `expire_time` and `reload_time` values.

A return value of zero indicates that `element_id` was successfully updated. A negative return values indicates an error occurred.

The update request input parameters correspond to the request function arguments.

```
<<dev realm timq param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevTimqUpdateInput,
    TIMQ_ElementID element_id ;
    TIMQ_TimeTicks expire_time ;
    TIMQ_TimeTicks reload_time ;
) ;
```

```
<<dev svc timq req: external definitions>>=
int
dev_timq_update(
    SVC_DevInstance timq,
    TIMQ_ElementID element_id,
    TIMQ_TimeTicks expire_time,
    TIMQ_TimeTicks reload_time)
{
    SVC_DevTimqUpdateInput input = {
        .block_size = sizeof(SVC_DevTimqUpdateInput),
        .element_id = element_id,
        .expire_time = expire_time,
        .reload_time = reload_time,
    } ;

    SVC_DevRequest timq_req = dev_req_encode(DEV_TIMQ_CLASS, timq_update, timq) ;
    return dev_realm_svc_call(timq_req, &input, NULL, NULL) ;
}
```

Because the timer queue is held in time relative order, one cannot just modify the queue element *in situ*. To maintain the correct time order of the timer queue, the foreground proxy function for updating a timer element must treat the update as the sequence:

stop, remove, insert, and start.

```
<<dev realm timq proxy: static function definitions>>=
static int
dev_realm_timq_update(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    TIMQ_ControlBlock *const tcb = &timer_queues[inst] ;
    rtcheck_not_NULL_return(tcb->notification.notify_proxy, -ERR_OPERATION_FAILED) ;

    SVC_DevTimqUpdateInput tq_input ;
    int result = copy_in_svc_param(input, sizeof(tq_input), &tq_input) ;
    rtcheck_zero_return(result, result) ;

    IRQn_Type irq_num = tcb->hdwr_controls.irq_num ;
    (void)stwd_begin_interrupt_section(irq_num) ;
//BEGIN INTERRUPT SECTION

    TIMQ_Element *element = blk_index_to_ref(tcb->element_allocator,
            tq_input.element_id) ;
    if (element != NULL) {
        timq_stop(tcb) ;
        timq_remove_element(tcb, element) ;

        element->expire_time = tq_input.expire_time ;
        element->reload_time = tq_input.reload_time ;           // ❶
        timq_insert_element(tcb, element) ;

        timq_start(tcb) ;
        result = 0 ;
    } else {
        result = -ERR_OPERATION_FAILED ;
    }

    stwd_end_interrupt_section(irq_num, true) ;
//END INTERRUPT SECTION

    return result ;
}
```

❶ *N.B.* the update request can change a one-shot into a periodic element and *vice versa*. It's not clear if that is a desirable side effect and application policy should determine if it is allowed or not.

Remaining Time for a Timer Element

```
<<dev svc timq req: external declarations>>=
extern int
dev_timq_remaining(
    SVC_DevInstance timq,
    TIMQ_ElementID element_id,
    TIMQ_TimeTicks *remaining) ;
```

timq
The instance number which identifies which timer queue which contains the element whose remaining time is obtained.

element_id
An identifier of a timer queue element. This value must be the same as some return value from a successful invocation of `dev_timq_insert`.

remaining
A pointer to an object of type `TIMQ_TimeTicks` where the remaining time is written by reference. A `NULL` value for `remaining` implies that no remaining time is returned. A `NULL` value may be used to determine if `element_id` exists in `timq` at all.

The `dev_timq_remaining` function obtains the number of time ticks which must elapse before a notification for `element_id` is sent.

A return value of 0 indicates that `element_id` was found and its remaining time was written to `remaining` (if the argument value was not `NULL`). A negative return value indicates failure.

```
<<dev realm timq param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevTimqRemainingInput,
    TIMQ_ElementID element_id ;
) ;
```

```
<<dev realm timq param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevTimqRemainingOutput,
    TIMQ_TimeTicks remaining ;
) ;
```

```
<<dev svc timq req: external definitions>>=
int
dev_timq_remaining(
    SVC_DevInstance timq,
    TIMQ_ElementID element_id,
    TIMQ_TimeTicks *remaining)
{
    SVC_DevTimqRemainingInput input = {
        .block_size = sizeof(SVC_DevTimqRemainingInput),
        .element_id = element_id,
    } ;
    SVC_DevTimqRemainingOutput output = {
        .block_size = sizeof(SVC_DevTimqRemainingOutput),
        .remaining = 0,
    } ;

    SVC_DevRequest timq_req = dev_req_encode(DEV_TIMQ_CLASS, timq_remaining, timq) ;
    int status = dev_realm_svc_call(timq_req, &input, &output, NULL) ;

    if (status == 0 && remaining != NULL) {
        *remaining = output.remaining ;
    }

    return status ;
```

```
}
```

Finally, the foreground proxy for determining the amount of remaining time wraps the queue operations of stop, remaining, and start.

```
<<dev realm timq proxy: static function definitions>>=
static int
dev_realm_timq_remaining(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    TIMQ_ControlBlock *const tcb = &timer_queues[inst] ;
    rtcheck_not_NULL_return(tcb->notification.notify_proxy, -ERR_OPERATION_FAILED) ;

    SVC_DevTimqRemainingInput tq_input ;
    int result = copy_in_svc_param(input, sizeof(tq_input), &tq_input) ;
    rtcheck_zero_return(result, result) ;

    IRQn_Type irq_num = tcb->hdwr_controls.irq_num ;
    (void)stwd_begin_interrupt_section(irq_num) ;
//BEGIN INTERRUPT SECTION

    TIMQ_Element *element = blk_index_to_ref(tcb->element_allocator,
            tq_input.element_id) ;
    if (element != NULL) {
        timq_stop(tcb) ;
        TIMQ_TimeTicks remaining = timq_remaining_time(tcb, element) ;
        timq_start(tcb) ;

        SVC_DevTimqRemainingOutput tq_output = {
            .block_size = sizeof(SVC_DevTimqRemainingOutput),
            .remaining = remaining,
        } ;
        result = copy_out_svc_result(output, sizeof(tq_output), &tq_output) ;
    } else {
        result = -ERR_OPERATION_FAILED ;
    }

    stwd_end_interrupt_section(irq_num, true) ;
//END INTERRUPT SECTION

    return result ;
}
```

Time Conversion

The time units used for the timer queue requests are device units. In this case the system timer device is configured to use unsigned UQ17.15 binary point numbers in units of seconds. The counter is clocked at 32 KiHz so that each timer tick represents approximately 30.5 μs. The maximum value for an UQ17.15 number is approximately 131,072 seconds or approximately 36.4 hours.

The following functions are useful for converting between device units and more conventional time units.

```
<<dev svc timq req: external declarations>>=
extern TIMQ_TimeTicks
dev_util_timq_ms_to_ticks(
    uint32_t ms) ;
```

ms
The number of milliseconds for which the equivalent timer queue device ticks is to be calculated.

The `dev_util_timq_ms_to_ticks` function converts a value in ms units to timer queue tick units. If the `ms` parameter is supplied with a value that would exceed the maximum allowed number of timer queue ticks, then the return value is silently truncated to the maximum supported by the timer queue.

```
<<dev svc timq req: external definitions>>=
TIMQ_TimeTicks
dev_util_timq_ms_to_ticks(
    uint32_t ms)
{
    uint64_t ticks =
            (((uint64_t)ms << 15) + UINT64_C(500)) / UINT64_C(1000) ; // ❶

    return (TIMQ_TimeTicks)min(ticks, UINT64_C(0xffffffff)) ;
}
```

❶ The arithmetic is done in 64 bits to avoid any overflow. The expression rounds the calculated millisecond value.

```
<<dev svc timq req: external declarations>>=
extern uint32_t
dev_util_timq_ticks_to_ms(
    TIMQ_TimeTicks ticks) ;
```

ticks
The number of timer queue device ticks to convert to ms.

The `dev_util_timq_ticks_to_ms` function converts time queue device ticks to ms.

```
<<dev svc timq req: external definitions>>=
uint32_t
dev_util_timq_ticks_to_ms(
    TIMQ_TimeTicks ticks)
{
    uint64_t msec =
            (((uint64_t)ticks * UINT64_C(1000)) +
            (uint64_t)btwd_mask(14, 0)) >> 15 ;    // ❶

    return (TIMQ_TimeTicks)msec ;
}
```

❶ Using 64-bit arithmetic with rounding.

It is possible to use the extra resolution beyond a millisecond which the timer queue frequency allows. Approximately 33 ticks is one ms. So, for example, timing 100 μs is approximately three ticks. As long as the 30.5 μs resolution is acceptable, it is possible to deal with shorter times in units of μs.

```
<<dev svc timq req: external declarations>>=
extern TIMQ_TimeTicks
dev_util_timq_us_to_ticks(
    uint32_t us) ;
```

us
A number of microseconds to convert to timer queue ticks.

The `dev_util_timq_us_to_ticks` function converts a microseconds value to timer queue device ticks.

```
<<dev svc timq req: external definitions>>=
TIMQ_TimeTicks
dev_util_timq_us_to_ticks(
    uint32_t us)
{
    uint64_t ticks = (((uint64_t)us << 15) + UINT64_C(500000)) / UINT64_C(1000000) ;

    return (TIMQ_TimeTicks)max(ticks, UINT64_C(1)) ;
}
```

Dispatching Timer Queue Requests

In keeping with the established pattern for device operations, timer queue requests have an additional level of dispatch and a function is required to find the foreground proxy function associated with the device operation.

```
<<dev realm timq proxy: external declarations>>=
extern int
dev_realm_timq(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

The implementation of the timer queue device realm dispatch function follows the same pattern used for other devices. It is a simple jump table held in an array.

```
<<dev realm timq proxy: external function definitions>>=
int
dev_realm_timq(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    int status = dev_req_validate(req, timq_operation_count, TIMQ_INSTANCES) ;
    rtcheck_zero_return(status, status) ;

    static const SVC_DevRequestProxy timq_proxies[timq_operation_count] = {
        [timq_open] = dev_realm_timq_open,
        [timq_close] = dev_realm_timq_close,
        [timq_insert] = dev_realm_timq_insert,
        [timq_remove] = dev_realm_timq_remove,
        [timq_update] = dev_realm_timq_update,
        [timq_remaining] = dev_realm_timq_remaining,
    } ;

    SVC_DevOperation timq_operation = dev_req_extract_operation(req) ;
```

```
    return timq_proxies[timq_operation](req, input, output, error) ;
}
```

The timer queue device is assigned a device class number and the `dev_realm_timq` function is placed into the jump table for device realm dispatch.

```
<<dev realm param: constants>>=
#define DEV_TIMQ_CLASS        1
```

```
<<svc entry: device request classes>>=
[DEV_TIMQ_CLASS] = dev_realm_timq,
```

Timer Compare Register IRQ Handling

The last part of timer queue operations is handling the interrupt requests from the system timer compare registers which time the queue elements. As stated previously, there is a separate IRQ vector for each compare register.

```
<<dev realm timq proxy: external function definitions>>=
void
STIMER_CMPR0_IRQHandler(void)
{
    timq_irq_handler(0) ;
}
```

```
<<dev realm timq proxy: external function definitions>>=
void
STIMER_CMPR1_IRQHandler(void)
{
    timq_irq_handler(1) ;
}
```

The code for handling the IRQ's is factored into a common function that is invoked with the timer queue instance number. In terms of basic queue operations, the IRQ handler is required to expire those elements which have timed out and restart the timer queue for any elements which remain in the queue.

```
<<dev realm timq proxy: static function definitions>>=
static void
timq_irq_handler(
    SVC_DevInstance inst)
{
    TIMQ_ControlBlock *const tcb = &timer_queues[inst] ;
    timq_expire_elements(tcb) ;                    // ❶
    timq_start(tcb) ;
}
```

❶ When the compare register interrupt happens, the queue element at the head of the queue will be zero. There is no possibility of preemption here by requests coming from the background since they run at the lower `SVC` exception priority. Given the flat IRQ priority scheme, there is no preemption from other interrupts. In other words, once here, running to completion is assured.

Code Layout

Following the pattern of code layout from Chapter 4, five files are created to hold the background request and foreground proxy code.

Timer Queue service requests

```
<<dev_svc_timq_req.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Function prototypes for timer queue service requests.
 *--
 */
#ifndef DEV_SVC_TIMQ_REQ_H_
#define DEV_SVC_TIMQ_REQ_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "dev_svc_req.h"
#include "dev_realm_timq_param.h"
/*
 * Data Type Declarations
 */
<<dev svc timq req: data type declarations>>
/*
 * External Declarations
 */
<<dev svc timq req: external declarations>>

#endif /* DEV_SVC_TIMQ_REQ_H_ */
```

Timer Queue Device Service Requests

```
<<dev_svc_timq_req.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Device Realm Timer Queue Request Implementation
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "useful.h"
#include "svc_req_errors.h"
#include "svc_req.h"
#include "dev_svc_req.h"
#include "dev_svc_timq_req.h"
```

```
#include "dev_realm_timq_param.h"
/*
 * External Functions
 */
<<dev svc timq req: external definitions>>
```

Device Realm Timer Queue Parameters

```
<<dev_realm_timq_param.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Parameter interface data structures for timer queue device requests.
 *--
 */
#ifndef DEV_REALM_TIMQ_PARAM_H_
#define DEV_REALM_TIMQ_PARAM_H_
/*
 * Include Files
 */
#include "dev_realm_param.h"
/*
 * Constants
 */
<<dev realm timq param: constants>>
/*
 * Data Type Declarations
 */
<<dev realm timq param: data type declarations>>

#endif /* DEV_REALM_TIMQ_PARAM_H_ */
```

Device Realm Timer Queue Proxy Header

```
<<dev_realm_timq_proxy.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Interfaces for device realm timq proxy functions.
 *--
 */
#ifndef DEV_REALM_TIMQ_PROXY_H_
#define DEV_REALM_TIMQ_PROXY_H_

/*
 * Include files
 */
#include "dev_realm_param.h"
```

```
/*
 * External Declarations
 */
<<dev realm timq proxy: external declarations>>

#endif /* DEV_REALM_TIMQ_PROXY_H_ */
```

```
<<svc handler: device include files>>=
#include "dev_realm_timq_proxy.h"
```

Device Realm Timer Queue Proxy Code Layout

```
<<dev_realm_timq_proxy.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for device realm timer queue proxy functions.
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "svc_req_errors.h"
#include "svc_proxy.h"
#include "dev_realm_proxy.h"
#include "dev_realm_timq_param.h"
#include "bg_req_queue.h"
#include "apollo3.h"
#include "bit_twiddle.h"
#include "sys_twiddle.h"
#include "useful.h"
#include "panic.h"
#include "rtcheck.h"
<<dev realm timq proxy: include files>>
/*
 * Constants
 */
<<dev realm timq proxy: constants>>
/*
 * Data Type Declarations
 */
<<dev realm timq proxy: data type declarations>>
/*
 * Forward References
 */
<<dev realm timq proxy: forward references>>
/*
 * Static Data Definitions
 */
<<dev realm timq proxy: static data definitions>>
/*
 * Static Inline Function Definitions
 */
```

```
<<dev realm timq proxy: static inline functions>>
/*
 * Static Function Definitions
 */
<<dev realm timq proxy: static function definitions>>
/*
 * External Function Definitions
 */
<<dev realm timq proxy: external function definitions>>
```

Testing

To test the timer queue functions, the Unity testing framework is used. The Unity framework is small, simple, pure "C", and can be used in low resource contexts.

Test execution synchronization

Before describing the test cases, a particular issue arises when testing code that is fundamentally asynchronous using a framework that assumes sequential, synchronous execution of the test cases. The Unity framework assumes that everything executes synchronously and it is only necessary to invoke the test function in order to run the test.

However, to complete a test may involve asynchronous execution. For example, if the test inserts an element in the timer queue, it must wait for the time to expire, the IRQ to run, the background notification to be posted, and the notification to be dispatched before the status of the test can be determined.

To accomplish the synchronization between the foreground and background, the `sys_ctrl_busy_wait` function, seen previously, is used. To use the busy/wait loop, a convention is used that passes a pointer to a boolean as the *closure* data for a device request and the notification function writes to the *terminate* variable via the closure pointer. This causes the loop to exit and the test appears to have simply executed sequential code.

For the test cases below, the timer queue notification function writes to the terminate variable by using a pointer to the terminate variable that was passed as the `SVC_DevNotifyClosure` value in the `dev_timq_insert` request. Recall that `SVC_DevNotifyClosure` is sized to insure that variables of that type can hold a pointer value.

First, the variable that is used in the synchronization must be defined.

```
<<timq-test: static data>>=
static bool volatile expiration_done ;
```

The background notification proxy function both records the notification information and writes to `expiration_done`.

```
<<timq-test: static data>>=
struct expiration {
    SVC_DevTimqNotification notify ;
    uint64_t time ;
} expired ;
```

```
<<timq-test: static functions>>=
static void
expired_proxy(
    SVC_DevTimqNotification const *const notify)
{
    expired.time = sys_eptime_ticks() ;
    expired.notify = *notify ;
    bool volatile *done_ref = (bool volatile *)notify->notify_closure ; // ❶
    *done_ref = true ;

    switch (notify->status) {
    case timq_expired:
```

```
        printf("timer element %d expired\n", notify->element_id) ;
        break ;

    case timq_reloaded:
        printf("timer element %d reloaded\n", notify->element_id) ;
        break ;

    case timq_discarded:
        printf("timer element %d closed\n", notify->element_id) ;
        break ;

    default:
        printf("unknown timer element status, %d\n", notify->status) ;
        break ;
    }
}
```

❶ This is where the closure value is used as a pointer to a boolean in order to synchronize to the sequentially running test case.

It is worth reiterating the execution sequence which synchronizes the execution of Unity tests with the asynchronous actions of timer queue elements. This technique is used in several places.

1. The `dev_timq_open` function establishes `expired_proxy` as the background proxy function. A pointer to `expired_prox` is placed with the notification when a timer element expires.
2. The `dev_timq_insert` function takes as an argument an item of *closure* data which is placed into the background notification when the inserted element expires. The value of the closure is the address of `expiration_done`, the variable which is monitored as a termination request.
3. A call to `sys_ctrl_busy_wait` enters an *event loop* where background notifications are dispatched. The `sys_ctrl_busy_` function checks the terminate variable, `expiration_done` in this case, to determine if one of the dispatched notification function changed its value to `true`.
4. If there are no notifications, the `sys_wait` function is used to put the processor to sleep.
5. When the system timer compare register matches, an IRQ is pended and the IRQ handler runs. When a timer element expires, the IRQ handler places a notification in the background notification queue. That notification contains the `expired_proxy` notification function pointer along with the closure data, which in this case is a pointer to `expiration_done` The interrupt causes the processor core to wake up and a notification queued to the background notification queue causes `sys_wait` to return.
6. Then, `sys_ctrl_busy_wait`, as part of its event loop, dispatches `expired_proxy`, which in turn sets `expiration_done` to true.
7. After `sys_ctrl_busy_wait` returns from dispatching the background notifications, it determines that the `expiration_don` variable has been changed to `true` and returns.

This may seem quite involved just for running a test case. However, this mechanism is at the heart of how background processing operates and can be used in an interactive situation to give the appearance that the test case function executed synchronously.

The Unity framework requires that `setUp` and `tearDown` functions be supplied. These functions are not used for this test.

```
<<timq-test: static functions>>=
void
setUp(void)
{
}
```

```
<<timq-test: static functions>>=
void
tearDown(void)
{
}
```

Test cases

Open timer queue

```
<<timq-test: static functions>>=
void
open_timer_queue(void)
{
    int status = dev_timq_open(0, expired_proxy) ;
    TEST_ASSERT(status == 0) ;
}
```

Remove element from empty timer queue

```
<<timq-test: static functions>>=
void
remove_from_empty(void) {
    TIMQ_ElementID elemID = dev_timq_remove(0, 0) ;
    TEST_ASSERT(elemID < 0) ;
}
```

Expire a timer element after 1 second

```
<<timq-test: static functions>>=
static void
one_sec_expire(void) {
    TIMQ_ElementID elemID = dev_timq_insert(0, dev_util_timq_ms_to_ticks(1000), 0,
            (SVC_DevNotifyClosure)&expiration_done) ;
    uint64_t insert_time = sys_eptime_ticks() ;
    TEST_ASSERT(elemID >= 0) ;

    sys_ctrl_busy_wait(&expiration_done) ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elemID == expired.notify.element_id) ;

    char ts_buf[32] ;
    sys_util_eptime_ticks_format(&insert_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d inserted: %s\n", elemID, ts_buf) ;
    sys_util_eptime_ticks_format(&expired.time, sizeof(ts_buf), ts_buf) ;
    printf("element %d expired: %s\n", expired.notify.element_id, ts_buf) ;

    uint32_t elapsed = dev_util_timq_ticks_to_ms(expired.time - insert_time) ;
    printf("elapsed time: %lu\n", elapsed) ;
    TEST_ASSERT(elapsed < 1005) ;
}
```

Insert a zero length expiration

```
<<timq-test: static functions>>=
void
zero_tick_expire(void) {
    TIMQ_ElementID elemID = dev_timq_insert(0, 0, 0,
            (SVC_DevNotifyClosure)&expiration_done) ;
    uint64_t insert_time = sys_eptime_ticks() ;
```

```
    TEST_ASSERT(elemID >= 0) ;

    sys_ctrl_busy_wait(&expiration_done) ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elemID == expired.notify.element_id) ;

    char ts_buf[32] ;
    sys_util_eptime_ticks_format(&insert_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d inserted: %s\n", elemID, ts_buf) ;
    sys_util_eptime_ticks_format(&expired.time, sizeof(ts_buf), ts_buf) ;
    printf("element %d expired: %s\n", expired.notify.element_id, ts_buf) ;

    uint32_t elapsed = dev_util_timq_ticks_to_ms(expired.time - insert_time) ;
    printf("elapsed time: %lu\n", elapsed) ;
    TEST_ASSERT(elapsed < 5) ;
}
```

Expire after 1 clock tick

```
<<timq-test: static functions>>=
void
one_tick_expire(void) {
    TIMQ_ElementID elemID = dev_timq_insert(0, 1, 0,
            (SVC_DevNotifyClosure)&expiration_done) ;
    uint64_t insert_time = sys_eptime_ticks() ;
    TEST_ASSERT(elemID >= 0) ;

    sys_ctrl_busy_wait(&expiration_done) ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elemID == expired.notify.element_id) ;

    char ts_buf[32] ;
    sys_util_eptime_ticks_format(&insert_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d inserted: %s\n", elemID, ts_buf) ;
    sys_util_eptime_ticks_format(&expired.time, sizeof(ts_buf), ts_buf) ;
    printf("element %d expired: %s\n", expired.notify.element_id, ts_buf) ;

    uint32_t elapsed = dev_util_timq_ticks_to_ms(expired.time - insert_time) ;
    printf("elapsed time: %lu\n", elapsed) ;
    TEST_ASSERT(elapsed < 5) ;
}
```

Insert larger time before smaller time

```
<<timq-test: static functions>>=
void
reorder_two_elements(void) {
    TIMQ_TimeTicks ticks_500 = dev_util_timq_ms_to_ticks(500) ;
    TEST_ASSERT(ticks_500 == 16384U) ;
    TIMQ_TimeTicks ticks_1000 = dev_util_timq_ms_to_ticks(1000) ;
    TEST_ASSERT(ticks_1000 == 32768U) ;

    TIMQ_ElementID elem_1000 = dev_timq_insert(0, ticks_1000, 0,
            (SVC_DevNotifyClosure)&expiration_done) ;
    uint64_t insert_time = sys_eptime_ticks() ;
    TIMQ_ElementID elem_500 = dev_timq_insert(0, ticks_500, 0,
            (SVC_DevNotifyClosure)&expiration_done) ;

    TEST_ASSERT(elem_1000 >= 0) ;
    TEST_ASSERT(elem_500 >= 0) ;

    sys_ctrl_busy_wait(&expiration_done) ;
```

```
    uint64_t elem_500_time = expired.time ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elem_500 == expired.notify.element_id) ;

    sys_ctrl_busy_wait(&expiration_done) ;
    uint64_t elem_1000_time = expired.time ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elem_1000 == expired.notify.element_id) ;

    char ts_buf[32] ;
    sys_util_eptime_ticks_format(&insert_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d inserted: %s\n", elem_1000, ts_buf) ;
    sys_util_eptime_ticks_format(&elem_500_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d expired: %s\n", elem_500, ts_buf) ;
    sys_util_eptime_ticks_format(&elem_1000_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d expired: %s\n", elem_1000, ts_buf) ;

    uint32_t elapsed = dev_util_timq_ticks_to_ms(elem_500_time - insert_time) ;
    printf("elem 500 time: %lu\n", elapsed) ;
    TEST_ASSERT(elapsed < 505) ;

    elapsed = dev_util_timq_ticks_to_ms(elem_1000_time - insert_time) ;
    printf("elem 1000 time: %lu\n", elapsed) ;
    TEST_ASSERT(elapsed < 1005) ;
}
```

Remaining time after one element has expired

```
<<timq-test: static functions>>=
void
remaining_two_elements(void) {
    TIMQ_TimeTicks ticks_250 = dev_util_timq_ms_to_ticks(250) ;
    TIMQ_TimeTicks ticks_1172 = dev_util_timq_ms_to_ticks(1172) ;

    TIMQ_ElementID elem_1172 = dev_timq_insert(0, ticks_1172, 0,
            (SVC_DevNotifyClosure)&expiration_done) ;
    uint64_t insert_time = sys_eptime_ticks() ;
    TIMQ_ElementID elem_250 = dev_timq_insert(0, ticks_250, 0,
            (SVC_DevNotifyClosure)&expiration_done) ;

    TEST_ASSERT(elem_1172 >= 0) ;
    TEST_ASSERT(elem_250 >= 0) ;

    sys_ctrl_busy_wait(&expiration_done) ;
    uint64_t elem_250_time = expired.time ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elem_250 == expired.notify.element_id) ;

    TIMQ_TimeTicks remaining ;
    int status = dev_timq_remaining(0, elem_1172, &remaining) ;
    TEST_ASSERT(0 == status) ;
    uint32_t remaining_ms = dev_util_timq_ticks_to_ms(remaining) ;

    sys_ctrl_busy_wait(&expiration_done) ;
    uint64_t elem_1172_time = expired.time ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elem_1172 == expired.notify.element_id) ;

    char ts_buf[32] ;
    sys_util_eptime_ticks_format(&insert_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d inserted: %s\n", elem_1172, ts_buf) ;
    sys_util_eptime_ticks_format(&elem_250_time, sizeof(ts_buf), ts_buf) ;
```

```
    printf("element %d expired: %s\n", elem_250, ts_buf) ;
    printf("remaining ms after 250 expired: %lu\n", remaining_ms) ;
    TEST_ASSERT(remaining_ms <= 922) ;

    sys_util_eptime_ticks_format(&elem_1172_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d expired: %s\n", elem_1172, ts_buf) ;

    uint32_t elapsed = dev_util_timq_ticks_to_ms(elem_250_time - insert_time) ;
    printf("elem 250 time: %lu\n", elapsed) ;
    TEST_ASSERT(elapsed < 255) ;

    elapsed = dev_util_timq_ticks_to_ms(elem_1172_time - insert_time) ;
    printf("elem 1172 time: %lu\n", elapsed) ;
    TEST_ASSERT(elapsed < 1177) ;
}
```

Update a larger expiration to be smaller than an existing one

```
<<timq-test: static functions>>=
void
update_second_element(void) {
    TIMQ_TimeTicks ticks_500 = dev_util_timq_ms_to_ticks(500) ;
    TIMQ_TimeTicks ticks_1000 = dev_util_timq_ms_to_ticks(1000) ;

    TIMQ_ElementID elem_1000 = dev_timq_insert(0, ticks_1000, 0,
            (SVC_DevNotifyClosure)&expiration_done) ;
    uint64_t insert_time_1000 = sys_eptime_ticks() ;

    TIMQ_ElementID elem_500 = dev_timq_insert(0, ticks_500, 0,
            (SVC_DevNotifyClosure)&expiration_done) ;
    uint64_t insert_time_500 = sys_eptime_ticks() ;

    TEST_ASSERT(elem_1000 >= 0) ;
    TEST_ASSERT(elem_500 >= 0) ;

    int status = dev_timq_update(0, elem_1000, dev_util_timq_ms_to_ticks(250), 0) ;
    TEST_ASSERT(status == 0) ;

    sys_ctrl_busy_wait(&expiration_done) ;
    uint64_t elem_1000_time = expired.time ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elem_1000 == expired.notify.element_id) ;

    sys_ctrl_busy_wait(&expiration_done) ;
    uint64_t elem_500_time = expired.time ;
    TEST_ASSERT_TRUE(expiration_done) ;
    TEST_ASSERT(elem_500 == expired.notify.element_id) ;

    char ts_buf[32] ;
    sys_util_eptime_ticks_format(&insert_time_1000, sizeof(ts_buf), ts_buf) ;
    printf("element %d inserted: %s\n", elem_1000, ts_buf) ;
    sys_util_eptime_ticks_format(&elem_1000_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d expired: %s\n", elem_1000, ts_buf) ;
    sys_util_eptime_ticks_format(&elem_500_time, sizeof(ts_buf), ts_buf) ;
    printf("element %d expired: %s\n", elem_500, ts_buf) ;

    uint32_t elapsed = dev_util_timq_ticks_to_ms(elem_1000_time - insert_time_1000) ;
    printf("elem 1000 time: %lu\n", elapsed) ;
    TEST_ASSERT(elapsed < 255) ;

    elapsed = dev_util_timq_ticks_to_ms(elem_500_time - insert_time_500) ;
    printf("elem 1000 time: %lu\n", elapsed) ;
```

```
    TEST_ASSERT(elapsed < 505) ;
}
```

A series of one second periodic expirations

```
<<timq-test: static functions>>=
void
one_sec_periodic(void) {
    TIMQ_TimeTicks ticks_1000 = dev_util_timq_ms_to_ticks(1000) ;
    TIMQ_ElementID elemID = dev_timq_insert(0, ticks_1000, ticks_1000,
            (SVC_DevNotifyClosure)&expiration_done) ;
    uint64_t insert_time = sys_eptime_ticks() ;
    TEST_ASSERT(elemID >= 0) ;

    for (unsigned i = 0 ; i < 4 ; i++) {
        sys_ctrl_busy_wait(&expiration_done) ;
        TEST_ASSERT(expiration_done) ;
        uint64_t expired_time = expired.time ;

        TIMQ_ElementID expired_element = expired.notify.element_id ;
        TEST_ASSERT(elemID == expired_element) ;

        char ts_buf[32] ;
        sys_util_eptime_ticks_format(&insert_time, sizeof(ts_buf), ts_buf) ;
        printf("element %d started: %s\n", expired_element, ts_buf) ;
        sys_util_eptime_ticks_format(&expired_time, sizeof(ts_buf), ts_buf) ;
        printf("element %d expired: %s\n", expired_element, ts_buf) ;

        uint32_t elapsed = dev_util_timq_ticks_to_ms(expired_time - insert_time) ;
        printf("elapsed time: %lu\n", elapsed) ;
        TEST_ASSERT(elapsed < 1005) ;

        insert_time = expired_time ;
    }
}
```

Close timer queue

```
<<timq-test: static functions>>=
void
close_timer_queue(void)
{
    int status = dev_timq_close(0) ;
    TEST_ASSERT(status == 0) ;
}
```

```
<<timq-test: external functions>>=
int
main(
    int argc,
    char **argv)
{
#       ifdef USE_SEGGER_RTT
    SEGGER_RTT_Init() ;
#       endif /* USE_SEGGER_RTT */

    printf("beginning test run\n") ;

    UNITY_BEGIN() ;

    RUN_TEST(open_timer_queue) ;
    RUN_TEST(remove_from_empty) ;
```

```
    RUN_TEST(one_sec_expire) ;
    RUN_TEST(zero_tick_expire) ;
    RUN_TEST(one_tick_expire) ;
    RUN_TEST(reorder_two_elements) ;
    RUN_TEST(remaining_two_elements) ;
    RUN_TEST(update_second_element) ;
    RUN_TEST(one_sec_periodic) ;
    RUN_TEST(close_timer_queue) ;

    return UNITY_END() ;
}
```

Timer Queue Tests Code Layout

```
<<time-mancery-timq-test.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Tests for timer queue interface.
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
#include <assert.h>
#include <stdio.h>
#include <inttypes.h>
#include <stdbool.h>
#include "sys_svc_req.h"
#include "dev_svc_timq_req.h"
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
#include "unity.h"
#include "bit_twiddle.h"
#include "fixedpoint.h"
/*
 * Constants
 */
<<timq-test: constants>>
/*
 * Data Type Declarations
 */
<<timq-test: data type declarations>>
/*
 * External Declarations
 */
<<timq-test: external declarations>>
/*
 * Forward References
 */
<<test util: forward references>>
<<timq-test: forward references>>
```

```
/*
 * Static Inline Functions
 */
<<timq-test: static inline functions>>
/*
 * External Inline Functions
 */
<<timq-test: external inline functions>>
/*
 * Static Data
 */
<<timq-test: static data>>
/*
 * Static Functions
 */
<<test util: static functions>>
<<timq-test: static functions>>
/*
 * External Functions
 */
<<timq-test: external functions>>
```

Epoch Time

Time is so important almost all computers have some means of counting the pulses of a fixed frequency clock. This can be used to keep track of calendar time by:

a. picking an arbitrary date, known as the *epoch*,

b. fixing that date to be the *zero* of time, and

c. incrementing a counter based on a fixed frequency clock pulse.

In this way, a counter, plus the implied side agreement as to the what calendar time was chosen for the epoch, yields a relatively simple way to keep track of calendar time by relating the count value back to conventional human time concepts. One common epoch date is, 1970-01-01 00:00:00+0000 (GMT), which is used in UNIX derived systems. The same epoch date is adopted in this design.

In this chapter, the system timer peripheral is revisited and the ability to track epoch time using the system timer as a counter is added. Code for its initialization was given in Chapter 2. The design strategy is to provide background requests to set and get the time of day in the form of the epoch count. When the count is set, the system timer peripheral is reset to zero. When the time is read, the system timer counter value is added to the last supplied epoch count. In other words, the system timer peripheral counter is used as an offset for the last set epoch time. This offset strategy allows for mapping between any differences in the units and resolution of the epoch count and the those of the system timer peripheral.

There is still a quandary. There is no authoritative source of the current time of day. That problem here is not solved here and it is assumed the current time is obtained from outside the system, perhaps using an external clock such as a GPS receiver or a battery backed RTC which has been previously set or even a WWVB clock. Given an epoch count from somewhere, the progress of time can be accurately tracked.

The interfaces to calendar time functions have been long established in the POSIX/UNIX-like world and they are also appropriate for this usage. Substitute implementations of some standard library functions that interface properly to the these specific time functions are provided.

Epoch time count

The system timer on the Apollo 3 survives normal resets and is only cleared at power up. Consequently, a place to store the base epoch time count that also survives reset is required. Fortunately, the `.noinit_bss` section, which was set aside previously, serves this purpose.

```
<<sys realm proxy: static data definitions>>=
static int64_t epoch_base __attribute__((section(".noinit_bss"))) ;
```

This value is typed as `int64_t` to match the current conventions used in the UNIX-like world and avoid any potential year "2038" problems.

Setting epoch time

To deal with epoch time, a POSIX-defined `struct timeval` value is used. This is not the most up-to-date approach to time in the larger computing world, but works well for these more limited purposes. A `struct timeval` value consists of a seconds member and a microseconds value. A `struct timeval` is defined having two members: a `time_t` number of seconds and a `suseconds_t` number of microseconds. The clock resolution is approximately 30.5 μs per tick, so the values will ultimately be truncated to that resolution.

```
<<sys realm param: include files>>=
#include <sys/time.h>
```

```
<<sys svc req: external declarations>>=
extern int
sys_eptime_set(
    struct timeval const *const tv) ;
```

tv
A pointer to an object of type `struct timeval` to which the current system epoch time is set.

The `sys_eptime_set` function sets the system epoch time to the value pointed to by `tv`. The function returns zero upon success and a negative value otherwise.

It is necessary to have an input argument to pass along the time value. Note the epoch time is passed by value and *not* simply as the function argument pointer value. This places the copy code in the background request rather than in the foreground proxy.

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysEptimeSetInput,
    struct timeval time ;
) ;
```

As usual, the background request is a veneer to marshal arguments for the service call.

```
<<sys svc req: external definitions>>=
int
sys_eptime_set(
    struct timeval const *const tv)
{
    rtcheck_not_NULL_return(tv, -ERR_INVALID_PARAM) ;

    SVC_SysEptimeSetInput input = {
        .block_size = sizeof(SVC_SysEptimeSetInput),
        .time = *tv,
    } ;

    return sys_realm_svc_call(SYS_EPTIME_SET, &input, NULL, NULL) ;
}
```

Since this is a system realm request, it is assigned a unique number and must be entered into the jump table of request functions.

```
<<sys realm param: constants>>=
#define SYS_EPTIME_SET    4
```

```
<<svc entry: system request functions>>=
[SYS_EPTIME_SET] = sys_realm_eptime_set,
```

The foreground proxy function has the usual interface.

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_eptime_set(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

The implementation of the request foreground proxy involves the following steps:

- Convert the `struct timeval` members into the internal representation of the system ticks counter. Recall that the system ticks value is a Q49.15 fixed binary point number in units of seconds.
- Store the converted ticks value into the `epoch_base` variable.
- Zero the system timer counter registers. In this way, the system timer counter becomes an offset to be added to `epoch_base` giving the current time.

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_eptime_set(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    (void)req ;
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_SysEptimeSetInput time_input ;
    int status = copy_in_svc_param(input, sizeof(time_input), &time_input) ;
    rtcheck_zero_return(status, status) ;

    int64_t frac_secs = (((int64_t)(time_input.time.tv_usec) << 15) + INT64_C(500000)) /
            INT64_C(1000000) ;
    epoch_base = ((int64_t)(time_input.time.tv_sec) << 15) + frac_secs ;

    BTWD_SET_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_FREEZE) ;
    BTWD_SET_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_CLEAR) ;
    BTWD_CLEAR_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_CLEAR) ;
    CTIMER->SNVR[0] = CTIMER->SNVR[1] = CTIMER->SNVR[2] = CTIMER->SNVR[3] = 0 ;      // ❶
    BTWD_CLEAR_REG_FIELD(&CTIMER->STCFG, CTIMER_STCFG_FREEZE) ;

    return 0 ;
}
```

❶ Clear out all the non-volatile RAM location that hold the overflow counts from the system timer even though only use one is used given the clock frequency of the system timer.

Getting epoch time

```
<<sys svc req: external declarations>>=
extern int
sys_eptime_get(
    struct timeval *const tv) ;
```

tv
A pointer to an object of type `struct timeval` to which the current system epoch time is stored.

The `sys_eptime_get` function gets the system epoch time and stores it into the value pointed to by `tv`. The function returns zero upon success and a negative value otherwise.

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysEptimeGetOutput,
    struct timeval time ;
) ;
```

```
<<sys svc req: external definitions>>=
int
sys_eptime_get(
    struct timeval *const tv)
{
    rtcheck_not_NULL_return(tv, -ERR_INVALID_PARAM) ;

    SVC_SysEptimeGetOutput output = {
        .block_size = sizeof(SVC_SysEptimeGetOutput),
        .time = {
            .tv_sec = 0,
            .tv_usec = 0,
        },
    } ;

    int status = sys_realm_svc_call(SYS_EPTIME_GET, NULL, &output, NULL) ;

    if (status == 0) {
        *tv = output.time ;
    }

    return status ;
}
```

```
<<sys realm param: constants>>=
#define SYS_EPTIME_GET    5
```

```
<<svc entry: system request functions>>=
[SYS_EPTIME_GET] = sys_realm_eptime_get,
```

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_eptime_get(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_eptime_get(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    (void)req ;
    assert(input == NULL) ; (void)input ;
    assert(error == NULL) ; (void)error ;

    uint64_t ts = dtwd_get_timestamp() + epoch_base ;

    SVC_SysEptimeGetOutput time_output = {
        .block_size = sizeof(time_output),
        .time.tv_sec = (int64_t)ts >> 15,
        .time.tv_usec = (((ts & (uint64_t)(btwd_mask(15, 0))) *
                UINT64_C(1000000)) + (uint64_t)btwd_mask(14, 0)) >> 15, // ❶
    } ;

    return copy_out_svc_result(output, sizeof(time_output), &time_output) ;
}
```

❶ The conversion of the timestamp from its Q49.15 representation to a microseconds count must:

a. Extract the fractional part using a 15-bit mask.

b. Multiply by 10^6 since there are a million μs per second.

c. Round the result by adding in a 14-bit mask before shifting off 15 bits to recover the integer number of microseconds represented by the fractional part of the timestamp value.

Getting high precision time ticks

For many purposes, the raw system timer value works well for timestamps. A direct interface to the system timer count value is provided. The is the best timer resolution provided in this system.

```
<<sys svc req: external declarations>>=
extern uint64_t
sys_eptime_ticks(void) ;
```

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysEptimeTicksOutput,
    uint64_t ticks ;
) ;
```

```
<<sys svc req: external definitions>>=
uint64_t
sys_eptime_ticks(void)
{
    assert(time != NULL) ;

    SVC_SysEptimeTicksOutput output = {
        .block_size = sizeof(output),
        .ticks = 0,
    } ;

    int status = sys_realm_svc_call(SYS_EPTIME_TICKS, NULL, &output, NULL) ;
```

```
    return status == 0 ? output.ticks : 0 ;
}
```

```
<<sys realm param: constants>>=
#define SYS_EPTIME_TICKS  6
```

```
<<svc entry: system request functions>>=
[SYS_EPTIME_TICKS] = sys_realm_eptime_ticks,
```

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_eptime_ticks(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_eptime_ticks(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    (void)req ;
    assert(input == NULL) ; (void)input ;
    assert(error == NULL) ; (void)error ;

    SVC_SysEptimeTicksOutput ticks_output = {
        .block_size = sizeof(ticks_output),
        .ticks = dtwd_get_timestamp(),
    } ;

    return copy_out_svc_result(output, sizeof(ticks_output), &ticks_output) ;
}
```

Formatting epoch ticks as timestamps

Raw ticks are sometimes needed as formatted values. A utility function is provided to perform the formatting.

```
<<sys svc req: external declarations>>=
extern int
sys_util_eptime_ticks_format(
    uint64_t const *const ticks_ref,
    unsigned buf_size,
    char buf[buf_size]) ;
```

ticks_ref
: A pointer to a ticks value as returned from the `sys_eptime_ticks` function. A `tics_ref` value of `NULL` indicates that the format function should invoke `sys_eptime_ticks` to obtain a value to format.

buf_size
: The length in bytes of the buffer given by `buf`.

buf
: A pointer to a character array which is `buf_size` in length.

The `sys_util_eptime_ticks_format` converts an epoch time ticks value into a human readable string. A ticks value is an unsigned UQ49.15 fixed radix point number in units of seconds. The returned string reports the number of seconds to three decimal places.

The return value of the function is the number of characters written to `buf` excluding the `NUL` terminating byte. The interpretation of the return value is the same as for `snprintf`. No more than `buf_size` bytes, including the `NUL` character are written to `buf`.

```
<<sys svc req: external definitions>>=
int
sys_util_eptime_ticks_format(
    uint64_t const *const ticks_ref,
    unsigned buf_size,
    char buf[buf_size])
{
    uint64_t ticks = ticks_ref == NULL ?
            sys_eptime_ticks() : *ticks_ref ;

    uint32_t ts_frac = (uint32_t)ticks & btwd_mask(15, 0) ;
    uint32_t frac_secs = ((ts_frac * UINT32_C(10000)) + btwd_mask(14, 0)) >> 15 ;
    uint32_t secs = ticks >> 15 ;                                          // ❶

    return snprintf(buf, buf_size, "%" PRIi32 ".%04" PRIu32 ": ", secs, frac_secs) ;
}
```

❶ *N.B.* the number of seconds is truncated to 32 bits in order to use a smaller version of libc which does not handle 64 bit numbers.

System Time Library

A number of standard function interfaces traditionally have been used in “C” programs. Here, implementations of those functions is supplied to aid in portability.

Time function

The standard interface to obtain the number of seconds since the epoch is the `time` function. The provided implementation uses the system realm requests previously shown.

The implementation is trivial and it is shown directly in its code file.

```
<<time.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation the time(2) function
 *--
 */

/*
 * Include files
 */
#include <time.h>
#include <sys/time.h>
#include "sys_svc_req.h"
/*
 * External Function Definitions
 */
time_t
time(
    time_t *t)
{
    struct timeval now ;
    int status = sys_eptime_get(&now) ;

    time_t now_secs = status < 0 ?
             (time_t)-1 : now.tv_sec ;

    if (t != NULL) {
        *t = now_secs ;
    }

    return now_secs ;
}
```

Get time of day

An implementation of the POSIX `gettimeofday` function is also provided. Note adopting current usage, the timezone field is considered obsolete.

```
<<gettimeofday.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation the gettimeofday(2) function
 *--
 */

/*
 * Include files
 */
```

```
#include <sys/time.h>
#include "sys_svc_req.h"
/*
 * External Function Definitions
 */
int
gettimeofday(
    struct timeval *restrict tv,
    void *restrict tz)
{
    int status = 0 ;

    if (tv != NULL) {                   // ❶
        status = sys_eptime_get(tv) ;
    }

    return status ;
}
```

❶ Since usage of `tz` is obsolete and its type is such that the pointer is not useful, it is not clear what invoking this function with a `NULL` value for `tv` is to accomplish. This is a simple attempt to be standards compliant.

Set time of day

```
<<settimeofday.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation the settimeofday(2) function
 *--
 */

/*
 * Include files
 */
#include <sys/time.h>
#include "sys_svc_req.h"
#include "svc_req_errors.h"
/*
 * External Function Definitions
 */
int
settimeofday(
    struct timeval const *tv,
    void *restrict tz)
{
    int status = 0 ;

    if (tv != NULL) {
        if (tv->tv_sec < 0 || tv->tv_usec < 0 || tv->tv_usec > 999999UL) {
            status = -ERR_INVALID_PARAM ;
        } else {
            status = sys_eptime_set(tv) ;
        }
    }
```

```
    return status ;
}
```

Testing

To test the Epoch time functions, test cases for the Unity testing framework are provided.

```
<<eptime-test: static functions>>=
void
setUp(void)
{
}
```

```
<<eptime-test: static functions>>=
void
tearDown(void)
{
}
```

Test cases

```
<<eptime-test: static functions>>=
static void
set_epoch_time(void) {
    struct tm tod = {
        .tm_sec = 0,
        .tm_min = 12,
        .tm_hour = 11,
        .tm_mday = 10,
        .tm_mon = 9,
        .tm_year = 2021 - 1900,
        .tm_wday = 5,
        .tm_yday = 0,
        .tm_isdst = 0,
    } ;

    struct timeval tv = {
        .tv_sec = mktime(&tod),
        .tv_usec = 0,
    } ;
    int status = settimeofday(&tv, NULL) ;
    TEST_ASSERT(status == 0) ;

    printf("time set: %s", asctime(&tod)) ;
}
```

```
<<eptime-test: static functions>>=
static void
get_epoch_time(void) {
    struct timeval tv ;

    int status = gettimeofday(&tv, NULL) ;
    TEST_ASSERT(status == 0) ;

    struct tm *tm = gmtime(&tv.tv_sec) ;
    printf("time get: %s", asctime(tm)) ;
}
```

```
<<eptime-test: static data>>=
static bool volatile timer_done ;
```

```
<<eptime-test: static functions>>=
static void
poll_epoch_time(void)
{
    TIMQ_ElementID elemID = dev_timq_insert(0,
            dev_util_timq_ms_to_ticks(1000),
            dev_util_timq_ms_to_ticks(1000),
            (SVC_DevNotifyClosure)&timer_done) ;

    TEST_ASSERT(elemID >= 0) ;
    for (unsigned i = 0 ; i < 10 ; i++) {
        sys_ctrl_busy_wait(&timer_done) ;

        struct timeval tv ;
        int status = gettimeofday(&tv, NULL) ;
        TEST_ASSERT(status == 0) ;

        struct tm *tm = gmtime(&tv.tv_sec) ;
        printf("time get: %s", asctime(tm)) ;
    }

    int status = dev_timq_remove(0, elemID) ;
    TEST_ASSERT(status == 0) ;
}
```

```
<<eptime-test: static functions>>=
static void
poll_epoch_ticks(void)
{
    TIMQ_ElementID elemID = dev_timq_insert(0,
            dev_util_timq_ms_to_ticks(1000),
            dev_util_timq_ms_to_ticks(1000),
            (SVC_DevNotifyClosure)&timer_done) ;

    TEST_ASSERT(elemID >= 0) ;
    for (unsigned i = 0 ; i < 10 ; i++) {
        sys_ctrl_busy_wait(&timer_done) ;

        char buf[64] ;
        sys_util_eptime_ticks_format(NULL, sizeof(buf), buf) ;
        printf("ticks: %s\n", buf) ;
    }

    int status = dev_timq_remove(0, elemID) ;
    TEST_ASSERT(status == 0) ;
}
```

```
<<eptime-test: external functions>>=
int
main(
    int argc,
    char **argv)
{
#       ifdef USE_SEGGER_RTT
    SEGGER_RTT_Init() ;
#       endif /* USE_SEGGER_RTT */
```

```
    printf("beginning Epoch time tests run\n") ;

    UNITY_BEGIN() ;

    int status = dev_timq_open(0, (SVC_DevTimqNotifyProxy)svc_proxy_var_sync) ;
    TEST_ASSERT(status == 0) ;

    RUN_TEST(set_epoch_time) ;
    RUN_TEST(get_epoch_time) ;
    RUN_TEST(poll_epoch_time) ;
    RUN_TEST(poll_epoch_ticks) ;

    status = dev_timq_close(0) ;
    TEST_ASSERT(status == 0) ;

    return UNITY_END() ;
}
```

Epoch time tests code layout

```
<<time-mancery-eptime-test.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Tests for epoch time and system time libraries.
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
#include <assert.h>
#include <stdio.h>
#include <inttypes.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>
#include "sys_svc_req.h"
#include "dev_svc_req.h"
#include "dev_svc_timq_req.h"
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
#include "unity.h"
/*
 * Constants
 */
<<eptime-test: constants>>
/*
 * Data Type Declarations
 */
<<eptime-test: data type declarations>>
/*
 * External Declarations
```

```
 */
<<eptime-test: external declarations>>
extern int settimeofday(struct timeval const *tv, void const *tz) ;
/*
 * Forward References
 */
<<test util: forward references>>
<<eptime-test: forward references>>
/*
 * Static Inline Functions
 */
<<eptime-test: static inline functions>>
/*
 * External Inline Functions
 */
<<eptime-test: external inline functions>>
/*
 * Static Data
 */
<<eptime-test: static data>>
/*
 * Static Functions
 */
<<test util: static functions>>
<<eptime-test: static functions>>
/*
 * External Functions
 */
<<eptime-test: external functions>>
```

Human Time

Humans have had a keen interest in measuring time long before possessing adequate technology to make precise measurements. The further from the equator one lives the more extensive the variation of the exposure to the Sun, making the adaptations required for survival and predicting the astronomical cycles essential.

By the twentieth century CE, technology advanced sufficiently to conquer time. The Earth's rotation around the Sun now can be tracked with such precision as to observe small variabilities in the period. Atomic clocks are accurate enough to observe relativistic time warp even at the slow speeds of aircraft (relative to the speed of light).

The periods of astronomical bodies such as the Earth and Moon are not convenient multiples of each other. For example, there are not an integral number of rotations of the Earth on its axis within one revolution around the Sun. The rotation of the Moon around the Earth also does not align evenly with the rotation of the Earth around the Sun.

Such complications have lead to a long history of intricate rules for keeping time in a manner to which humans have become accustomed. Humans are adept at keeping track of such rules. For computers, keeping time in a manner convenient to human rules is a "fussy" undertaking. It is not uncommon for modern microcontrollers to have specialized peripherals which count time and perform the some of the modulus arithmetic associated with day, months, years, etc.

Overview of the RTC

The Apollo 3 has a Real Time Clock (RTC) peripheral which is capable of keeping track of time in human oriented units as well as signaling when the current time matches an alarm time. The Apollo 3 data sheet describes the details of the peripheral register interface. The RTC features used are:

- The RTC keeps time in a *broken out* format, *i.e.* there are separate register fields for each time component such as hour, minute, etc.

- The RTC is set to run in the 21st century only.
- The RTC has an alarm function which can signal a match for one given date and which can repeatedly signal matches on selected date fields. Access to the alarm and its repeat function are provided.
- The hours counter is run in so called, *24 hour*, mode as opposed to a 12 hour mode which uses an additional bit for AM or PM.

BCD Conversion

As is common for RTC of peripherals[3], the time values in the peripheral registers are in Binary Coded Decimal (BCD) format. It is more convenient, outside of the device specific code, to deal with conventional 2's complement binary numbers. A couple of functions are required to perform the conversions.

```
<<dev realm rtc proxy: static inline functions>>=
static inline unsigned
bin_to_bcd(
    uint8_t bin)
{
    assert(bin <= 99U) ;                          // ❶

    return ((bin / 10U) << 4U) | (bin % 10U) ;
}
```

❶ One byte can hold two BCD digits in the usual encoding. Interestingly, there are several ways to encode decimal numbers as a binary byte.

```
<<dev realm rtc proxy: static inline functions>>=
#include <stdio.h>
static inline unsigned
bcd_to_bin(
    uint8_t bcd)
{
    return ((bcd >> 4) * 10U) + (bcd & btwd_mask(4, 0)) ;
}
```

RTC time value

The following structure is used to hold the values for the broken out time used by the RTC.

```
<<dev realm rtc param: data type declarations>>=
typedef struct {
    uint16_t year ;
    uint8_t month ;
    uint8_t date ;
    uint8_t hour ;
    uint8_t minute ;
    uint8_t second ;
    uint8_t hundredth ;
    uint8_t weekday ;
} RTC_Time ;
```

The structure members correspond directly to the RTC peripheral register values, except for the `year` member. When dealing with an RTC time value in the application, a complete year value in the current era (CE) is used, *e.g.* 2022. In the RTC peripheral registers, there is only a single byte of BCD encoded year (which can hold 0 to 99) and the *century* is indicated by a separate bit. This use of the peripheral is only concerned with keeping time in the 2000's CE and the century bit is set accordingly. Before writing the year into the RTC register, century portion is subtracted off.

[3] Undoubtedly for some backwards compatible reason.

```
<<dev realm rtc proxy: constants>>=
#define RTC_CENTURY_BASE    2000U
```

The following function converts the time representation from "engineering" units (*i.e.* 2's complement with the century included) to "device" units (*i.e.* BCD values with the century removed). As part of the conversion, validation is done. This conversion is used when the RTC time value is set into the peripheral registers. *N.B.* there does not seem to be anywhere in the Apollo 3 data sheet where the valid ranges of the time fields is specified. The validation ranges below were determined empirically.

```
<<dev realm rtc proxy: static function definitions>>=
static int
eng_time_to_dev_time(
    RTC_Time const *const eng_time,
    RTC_Time *const dev_time)
{
    rtcheck_range_return(eng_time->year, RTC_CENTURY_BASE, RTC_CENTURY_BASE + 100,
            -ERR_INVALID_PARAM) ;
    dev_time->year = bin_to_bcd(eng_time->year - RTC_CENTURY_BASE) ;

    rtcheck_range_return(eng_time->month, 1, 13, -ERR_INVALID_PARAM) ;
    dev_time->month = bin_to_bcd(eng_time->month) ;

    rtcheck_range_return(eng_time->date, 1, 32, -ERR_INVALID_PARAM) ;
    dev_time->date = bin_to_bcd(eng_time->date) ;

    rtcheck_max_return(eng_time->hour, 24, -ERR_INVALID_PARAM) ;
    dev_time->hour = bin_to_bcd(eng_time->hour) ;

    rtcheck_max_return(eng_time->minute, 60, -ERR_INVALID_PARAM) ;
    dev_time->minute = bin_to_bcd(eng_time->minute) ;

    rtcheck_max_return(eng_time->second, 60, -ERR_INVALID_PARAM) ;
    dev_time->second = bin_to_bcd(eng_time->second) ;

    rtcheck_max_return(eng_time->hundredth, 100, -ERR_INVALID_PARAM) ;
    dev_time->hundredth = bin_to_bcd(eng_time->hundredth) ;

    rtcheck_max_return(eng_time->weekday, 7, -ERR_INVALID_PARAM) ;
    dev_time->weekday = bin_to_bcd(eng_time->weekday) ;

    return 0 ;
}
```

The inverse conversion is somewhat simpler and used when the RTC time value is read.

```
<<dev realm rtc proxy: static function definitions>>=
static void
dev_time_to_eng_time(
    RTC_Time const *const dev_time,
    RTC_Time *const eng_time)
{
    eng_time->year = bcd_to_bin(dev_time->year) + RTC_CENTURY_BASE ;
    eng_time->month = bcd_to_bin(dev_time->month) ;
    eng_time->date = bcd_to_bin(dev_time->date) ;
    eng_time->hour = bcd_to_bin(dev_time->hour) ;
    eng_time->minute = bcd_to_bin(dev_time->minute) ;
    eng_time->second = bcd_to_bin(dev_time->second) ;
    eng_time->hundredth = bcd_to_bin(dev_time->hundredth) ;
    eng_time->weekday = bcd_to_bin(dev_time->weekday) ;
}
```

Reading the RTC time

The following function reads the RTC time from the hardware and returns the time in engineering units. Because the peripheral packs several components of the time into a single register, the code contains some bit twiddling to extract the fields from the register values.

```
<<dev realm rtc proxy: static function definitions>>=
static int
rtc_read_time(
    RTC_ControlBlock const *const rcb,
    RTC_Time *const rtc_time)
{
    assert(rcb != NULL) ; (void)rcb ;
    assert(rtc_time != NULL) ;

    uint32_t lower ;
    uint32_t upper ;
    bool did_read = dtwd_get_rtc_counters(&lower, &upper) ;             // ❶
    rtcheck_return(did_read, -ERR_OPERATION_FAILED) ;

    RTC_Time dev_time ;

    dev_time.weekday = BTWD_FIELD_EXTRACT(upper, RTC_CTRUP_CTRWKDY) ;
    dev_time.year = BTWD_FIELD_EXTRACT(upper, RTC_CTRUP_CTRYR) ;
    dev_time.month = BTWD_FIELD_EXTRACT(upper, RTC_CTRUP_CTRMO) ;
    dev_time.date = BTWD_FIELD_EXTRACT(upper, RTC_CTRUP_CTRDATE) ;
    dev_time.hour = BTWD_FIELD_EXTRACT(lower, RTC_CTRLOW_CTRHR) ;
    dev_time.minute = BTWD_FIELD_EXTRACT(lower, RTC_CTRLOW_CTRMIN) ;
    dev_time.second = BTWD_FIELD_EXTRACT(lower, RTC_CTRLOW_CTRSEC) ;
    dev_time.hundredth = BTWD_FIELD_EXTRACT(lower, RTC_CTRLOW_CTR100) ;

    dev_time_to_eng_time(&dev_time, rtc_time) ;

    return 0 ;
}
```

❶ Because there are retries involved, reading the RTC counters is factored into the device register twiddling code.

Updating the RTC time

The following function updates the time in the RTC peripheral. Time in this case is considered to be in engineering units and is converted to device units before inserting the various fields into the peripheral registers. Again, bit twiddling to pack the peripheral registers is necessary.

```
<<dev realm rtc proxy: static function definitions>>=
static int
rtc_update_time(
    RTC_ControlBlock const *const rcb,
    RTC_Time const *const rtc_time)
{
    assert(rcb != NULL) ;
    assert(rtc_time != NULL) ;

    RTC_Type *const rtc = rcb->hdwr_controls.rtc_regs ;

    RTC_Time dev_time ;
    int converted = eng_time_to_dev_time(rtc_time, &dev_time) ;
    rtcheck_zero_return(converted, converted) ;

    uint32_t ctrlow = rtc->CTRLOW ;
```

```
    ctrlow = BTWD_FIELD_INSERT(ctrlow, dev_time.hundredth, RTC_CTRLOW_CTR100) ;
    ctrlow = BTWD_FIELD_INSERT(ctrlow, dev_time.second, RTC_CTRLOW_CTRSEC) ;
    ctrlow = BTWD_FIELD_INSERT(ctrlow, dev_time.minute, RTC_CTRLOW_CTRMIN) ;
    ctrlow = BTWD_FIELD_INSERT(ctrlow, dev_time.hour, RTC_CTRLOW_CTRHR) ;

    uint32_t ctrup = rtc->CTRUP ;
    ctrup = BTWD_FIELD_INSERT(ctrup, dev_time.weekday, RTC_CTRUP_CTRWKDY) ;
    ctrup = BTWD_FIELD_INSERT(ctrup, dev_time.year, RTC_CTRUP_CTRYR) ;
    ctrup = BTWD_FIELD_INSERT(ctrup, dev_time.month, RTC_CTRUP_CTRMO) ;
    ctrup = BTWD_FIELD_INSERT(ctrup, dev_time.date, RTC_CTRUP_CTRDATE) ;

    BTWD_SET_REG_FIELD(&rtc->RTCCTL, RTC_RTCCTL_WRTC) ;          // ❶
    rtc->CTRLOW = ctrlow ;
    rtc->CTRUP = ctrup ;
    BTWD_CLEAR_REG_FIELD(&rtc->RTCCTL, RTC_RTCCTL_WRTC) ;

    return 0 ;
}
```

❶ *N.B.* it is important to insure that the RTC is not stopped when attempting to write to the counter registers (*i.e.* the `RSTOP` bit of `RTCCTL` must be 0). As determined empirically, if the RTC is stopped, you can't write to the counter registers regardless of the value of the `WRTC` bit. Seem that the `RSTOP` is intended for "stop watch" style operations. There is an allusion to this in the data sheet. Also, the `WRTC` only controls the ability to write to `CTRLOW` and `CTRUP`. Other RTC registers don't seem to be affected by the `WRTC` setting. The data sheet is vague in this area.

RTC alarm value

The RTC has compare registers that can generate an interrupt when they match the counter values of the RTC. Only a single set of alarm registers is provided by the RTC and so Only a single alarm setting at a time is supported.

The alarm generates repeating alarms. It is either disabled, or generates a periodic interrupt. The repeat function controls how many alarm registers are considered in determining if the current time matches the alarm.

```
<<dev realm rtc param: data type declarations>>=
typedef enum {
    alarm_disabled = 0,
    alarm_every_year,
    alarm_every_month,
    alarm_every_week,
    alarm_every_day,
    alarm_every_hour,
    alarm_every_minute,
    alarm_every_second,
} RTC_AlarmRepeat ;
```

The broken out structure for an alarm is similar to that of the RTC counters. Note there is no `year` member. The alarm operates by matching all the counter values that are more frequent than the selected alarm repeat.

```
<<dev realm rtc param: data type declarations>>=
typedef struct {
    RTC_AlarmRepeat repeat ;
    uint8_t month ;
    uint8_t date ;
    uint8_t hour ;
    uint8_t minute ;
    uint8_t second ;
    uint8_t hundredth ;
    uint8_t weekday ;
} RTC_Alarm ;
```

The alarm comparison match registers are also held in BCD encoding.

```
<<dev realm rtc proxy: static function definitions>>=
static int
eng_alarm_to_dev_alarm(
    RTC_Alarm const *const eng_time,
    RTC_Alarm *const dev_time)
{
    dev_time->repeat = eng_time->repeat ;
    dev_time->month = bin_to_bcd(eng_time->month) ;
    dev_time->date = bin_to_bcd(eng_time->date) ;
    dev_time->hour = bin_to_bcd(eng_time->hour) ;
    dev_time->minute = bin_to_bcd(eng_time->minute) ;
    dev_time->second = bin_to_bcd(eng_time->second) ;
    dev_time->hundredth = bin_to_bcd(eng_time->hundredth) ;
    dev_time->weekday = bin_to_bcd(eng_time->weekday) ;

    return 0 ;
}
```

Updating the Alarm time

The following function updates the alarm time in the RTC peripheral. Time in this case is considered to be in engineering units and is converted to device units before inserting the various fields into the peripheral registers. Like the RTC time fields, the alarm values are inserted into bit fields contained in two peripheral registers.

```
<<dev realm rtc proxy: static function definitions>>=
static int
rtc_update_alarm(
    RTC_ControlBlock const *const rcb,
    RTC_Alarm const *const alarm_time)
{
    assert(rcb != NULL) ;
    assert(alarm_time != NULL) ;

    RTC_Type *const rtc = rcb->hdwr_controls.rtc_regs ;

    RTC_Alarm dev_alarm ;
    int converted = eng_alarm_to_dev_alarm(alarm_time, &dev_alarm) ;
    rtcheck_zero_return(converted, converted) ;

    uint32_t almlow = 0 ;
    almlow = BTWD_FIELD_INSERT(almlow, dev_alarm.hundredth, RTC_ALMLOW_ALM100) ;
    almlow = BTWD_FIELD_INSERT(almlow, dev_alarm.second, RTC_ALMLOW_ALMSEC) ;
    almlow = BTWD_FIELD_INSERT(almlow, dev_alarm.minute, RTC_ALMLOW_ALMMIN) ;
    almlow = BTWD_FIELD_INSERT(almlow, dev_alarm.hour, RTC_ALMLOW_ALMHR) ;
    rtc->ALMLOW = almlow ;

    uint32_t almup = 0 ;
    almup = BTWD_FIELD_INSERT(almup, dev_alarm.weekday, RTC_ALMUP_ALMWKDY) ;
    almup = BTWD_FIELD_INSERT(almup, dev_alarm.month, RTC_ALMUP_ALMMO) ;
    almup = BTWD_FIELD_INSERT(almup, dev_alarm.date, RTC_ALMUP_ALMDATE) ;
    RTC->ALMUP = almup ;

    uint32_t rtcctl = rtc->RTCCTL ;
    rtcctl = BTWD_FIELD_INSERT(rtcctl, dev_alarm.repeat, RTC_RTCCTL_RPT) ;
    rtc->RTCCTL = rtcctl ;

    return 0 ;
}
```

RTC Requests

In this section, device interface and control code for the RTC device is shown. The following diagram shows the peripherals involved in the RTC device code.

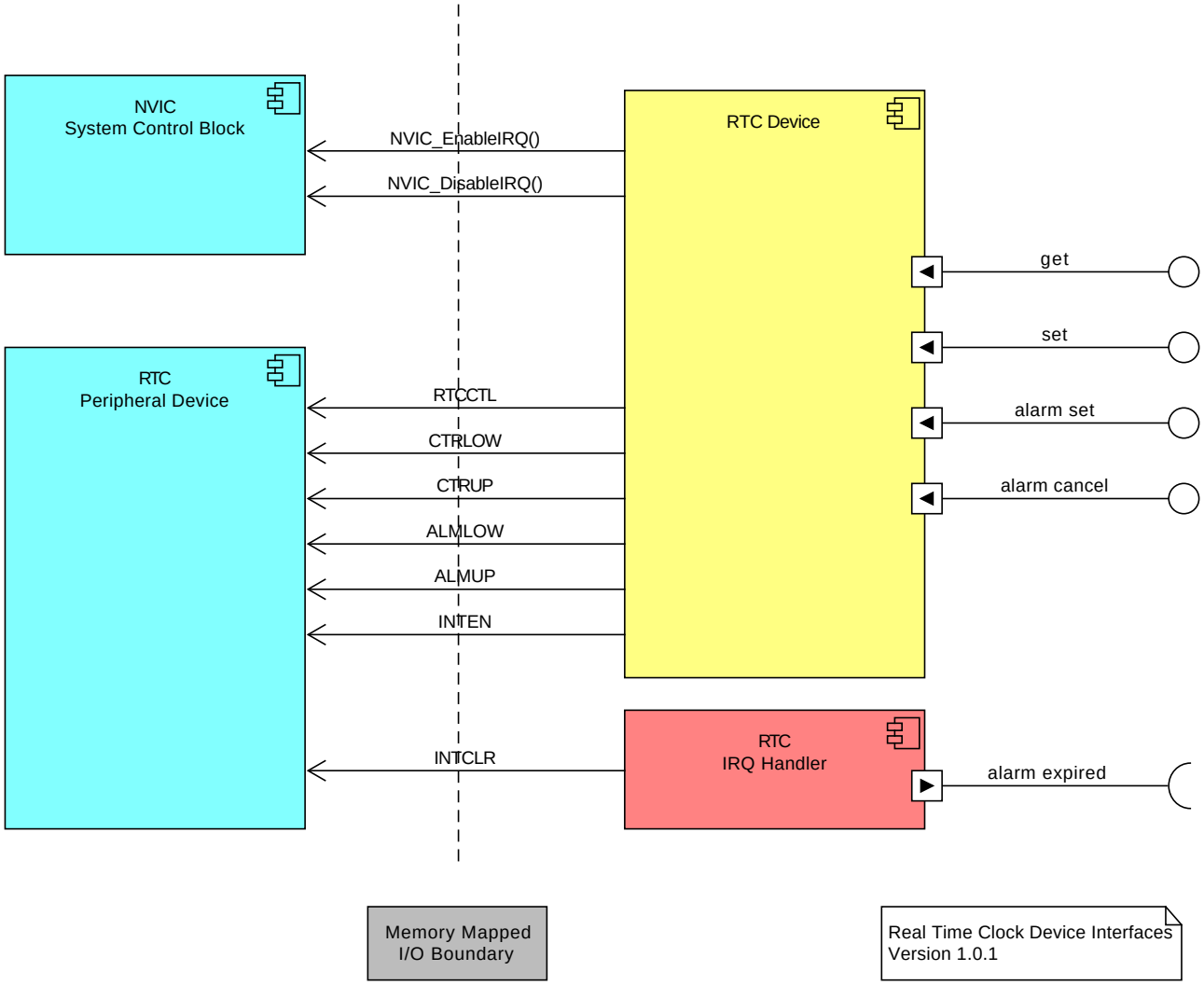

Figure 7.4: RTC Interface Components

RTC request encoding

An encoding for the operations supported by the RTC device is given by the following enumeration.

```
<<dev realm rtc param: data type declarations>>=
typedef enum {
    rtc_set,
    rtc_get,
    rtc_alarm_set,
    rtc_alarm_cancel,

    rtc_operation_count     // last
} SVC_rtcRequest ;
```

Background Notification

Since the alarming capability of the RTC is interrupt driven, the structure of the background notification which is sent when the RTC signals the alarm must be defined. The background notification carries a status value to indicate what happened.

```
<<dev realm rtc param: data type declarations>>=
typedef enum {
    alarm_expired = 3,                  // ❶
    alarm_discarded,
} RTC_AlarmStatus ;
```

❶ A lot of zeros float around in code. Start at something which has a lower probability of being a garbage memory value. This encoding also avoids some of the implicit “truthy-ness” which, in “C”, can arise when integer values are used in a boolean context.

The `alarm_expired` status indicates that the alarm values match the current time as governed by the alarm repeat control. The `alarm_discarded` status indicates that a previous alarm request was discarded because the time value in the peripheral was changed by a `rtc_set` operation or that a new alarm was set by the `rtc_set_alarm` operation. Changing the time in the RTC causes any pending alarm request to be canceled automatically. Setting a new alarm when there is one already pending cancels the previous alarm. The `alarm_discarded` status indicates to background processing that the pending alarm will never be received.

```
<<dev realm rtc param: data type declarations>>=
DECLARE_DEV_NOTIFICATION(SVC_DevRTCNotification,
    RTC_AlarmStatus status ;
    RTC_Time time ;
) ;
```

Note that in addition to the alarm status, the current RTC time when the alarm expired is also included in the background notification as a convenience.

The background notification proxy takes as an argument the device specific form of the notification.

```
<<dev realm rtc param: data type declarations>>=
typedef void (*SVC_DevRTCNotifyProxy)(SVC_DevRTCNotification const *const params) ;
```

The code to send the RTC background notfication is factored into a single function.

```
<<dev realm rtc proxy: static function definitions>>=
static void
send_rtc_notification(
    RTC_ControlBlock *const rcb,
    RTC_AlarmStatus status)
{
    assert(rcb != NULL) ;
    if (rcb->notification.notify_proxy == NULL) {
        return ;
    }

    int result = rtc_read_time(rcb, &rcb->notification.time) ;
    rtcheck(result == 0) ;

    rcb->notification.status = status ;
    bool sent = send_bg_notification(&rcb->notification) ;
    rtcheck(sent) ;
}
```

There is only one RTC peripheral in the Apollo 3 SOC.

```
<<dev realm rtc param: constants>>=
#define RTC_INSTANCES   1
```

Even though there is only one RTC peripheral, it is convenient to associate its IRQ number to the peripheral registers address block.

```
<<dev realm rtc proxy: data type declarations>>=
typedef struct {
    IRQn_Type irq_num ;
    RTC_Type *rtc_regs ;
} RTC_HdwrControls ;
```

irq_num
The interrupt number associated with the RTC compare register.

rtc_regs
A pointer to the memory mapped I/O for the RTC registers.

Following the established pattern, a data structure is defined to hold all the required control information for the RTC.

```
<<dev realm rtc proxy: data type declarations>>=
typedef struct {
    SVC_DevRTCNotification notification ;
    SVC_DevRTCNotifyProxy notify_proxy ;
    SVC_DevNotifyClosure notify_closure ;
    RTC_HdwrControls hdwr_controls ;
} RTC_ControlBlock ;
```

The control block is initialized at compile time and is ready for use when the system starts.

```
<<dev realm rtc proxy: static data definitions>>=
static RTC_ControlBlock rtcs[RTC_INSTANCES] = {
    [0] = {
        .notification = {
            .block_size = sizeof(SVC_DevRTCNotification),
            .notify_proxy = NULL,
            .notify_closure = 0,
            .device_class = DEV_RTC_CLASS,
            .device_instance = 0,
            .time = {
                .year = 0,
                .month = 0,
                .date = 0,
                .hour = 0,
                .minute = 0,
                .second = 0,
                .hundredth = 0,
                .weekday = 0,
            },
            .status = alarm_discarded,
        },
        .hdwr_controls = {
            .irq_num = RTC_IRQn,
            .rtc_regs = RTC,
        }
    },
} ;
```

Setting the time of day

In this section and the following, the usual pattern of functions for device control are given for the RTC case. First, the interface to background processing and then the foreground proxy function which runs with privilege and access the peripheral device.

```
<<dev svc rtc req: external declarations>>=
extern int
dev_rtc_set(
    SVC_DevInstance rtc,
    RTC_Time const *const time) ;
```

rtc
The instance number of the RTC peripheral which is to be set.

time
A pointer to a broken out time to which the RTC is to be set.

The `dev_rtc_set` function sets the Real Time Clock (RTC) instance given by, `rtc`, to the time value pointed to by, `time`. A return value of 0 indicates success and negative values indicate an error.

Following the usual pattern, an input structure is defined to pass the time argument to the foreground proxy.

```
<<dev realm rtc param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevRTCSetInput,
    RTC_Time time ;
) ;
```

```
<<dev svc rtc req: external definitions>>=
int
dev_rtc_set(
    SVC_DevInstance rtc,
    RTC_Time const *const time)
{
    SVC_DevRTCSetInput input = {
        .block_size = sizeof(SVC_DevRTCSetInput),
        .time = *time,
    } ;

    SVC_DevRequest rtc_req = dev_req_encode(DEV_RTC_CLASS, rtc_set, rtc) ;

    return dev_realm_svc_call(rtc_req, &input, NULL, NULL) ;
}
```

Since the RTC is run in a particular configuration, *i.e.* 24 hour mode, 20ˆth century, etc. the configuration is set each time the RTC time is set. This insures the peripheral is operated as this code is expecting. Again, note it is *not* the purpose to try to build a completely general interface which supports every possible configuration of the peripheral device.

```
<<dev realm rtc proxy: static function definitions>>=
static int
dev_realm_rtc_set(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    RTC_ControlBlock *const rcb = &rtcs[inst] ;
    RTC_Type *const rtc = rcb->hdwr_controls.rtc_regs ;

    NVIC_DisableIRQ(rcb->hdwr_controls.irq_num) ;
```

```
    BTWD_CLEAR_REG_FIELD(&rtc->INTEN, RTC_INTEN_ALM) ;
    rtc->INTCLR = RTC_INTCLR_ALM_Msk ;

    uint32_t ctrl_value = RTC->RTCCTL ;
    ctrl_value = BTWD_FIELD_CLEAR(ctrl_value, RTC_RTCCTL_HR1224) ;
    ctrl_value = BTWD_FIELD_INSERT(ctrl_value, 0, RTC_RTCCTL_RPT) ;
    RTC->RTCCTL = ctrl_value ;

    send_rtc_notification(rcb, alarm_discarded) ;
    rcb->notify_proxy = NULL ;

    SVC_DevRTCSetInput rtc_input ;
    int status = copy_in_svc_param(input, sizeof(rtc_input), &rtc_input) ;
    rtcheck_zero_return(status, status) ;

    return rtc_update_time(rcb, &rtc_input.time) ;
}
```

Getting the time of day

```
<<dev svc rtc req: external declarations>>=
extern int
dev_rtc_get(
    SVC_DevInstance rtc,
    RTC_Time *const time) ;
```

rtc
The instance number of the RTC peripheral which is to be accessed.

time
A pointer to a broken out time which is used to return the current time of day.

The `dev_rtc_get` function obtains the time of day from the Real Time Clock (RTC) instance given by, `rtc`, and returns the time into the memory object pointed to by, `time`. A return value of 0 indicates success and negative values indicate an error.

```
<<dev realm rtc param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevRTCGetOutput,
    RTC_Time time ;
) ;
```

```
<<dev svc rtc req: external definitions>>=
int
dev_rtc_get(
    SVC_DevInstance rtc,
    RTC_Time *const time)
{
    SVC_DevRTCGetOutput output = {
        .block_size = sizeof(SVC_DevRTCGetOutput),
    } ;

    SVC_DevRequest rtc_req = dev_req_encode(DEV_RTC_CLASS, rtc_get, rtc) ;

    int status = dev_realm_svc_call(rtc_req, NULL, &output, NULL) ;

    if (status == 0) {
        *time = output.time ;
```

```
    }

    return status ;
}
```

```
<<dev realm rtc proxy: static function definitions>>=
static int
dev_realm_rtc_get(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(input == NULL) ; (void)input ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    RTC_ControlBlock *rcb = &rtcs[inst] ;

    SVC_DevRTCGetOutput rtc_output = {
        .block_size = sizeof(rtc_output),
    } ;

    int status = rtc_read_time(rcb, &rtc_output.time) ;
    if (status == 0) {
        status = copy_out_svc_result(output, sizeof(rtc_output), &rtc_output) ;
    }

    return status ;
}
```

Setting an alarm

```
<<dev svc rtc req: external declarations>>=
extern int
dev_rtc_alarm_set(
    SVC_DevInstance rtc,
    RTC_Alarm const *const alarm,
    SVC_DevRTCNotifyProxy notify_proxy,
    SVC_DevNotifyClosure notify_closure) ;
```

rtc
: The instance number of the RTC peripheral on which the alarm is to be set.

alarm
: A pointer to a broken out alarm time which is to be set.

notify_proxy
: A pointer to a background notification function which is passed through in the notification.

notify_closure
: User specified data value returned in the alarm notification.

The `dev_rtc_alarm_set` function sets an alarm specified by the object pointed to by, `alarm`, on the RTC given by, `rtc`. When the alarm expires, a background notification is queued. A return value of 0 indicates success and negative values indicate an error.

```
<<dev realm rtc param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevRTCAlarmSetInput,
    RTC_Alarm alarm ;
    SVC_DevRTCNotifyProxy notify_proxy ;
    SVC_DevNotifyClosure notify_closure ;
) ;
```

```
<<dev svc rtc req: external definitions>>=
int
dev_rtc_alarm_set(
    SVC_DevInstance rtc,
    RTC_Alarm const *const alarm,
    SVC_DevRTCNotifyProxy notify_proxy,
    SVC_DevNotifyClosure notify_closure)
{
    SVC_DevRTCAlarmSetInput input = {
        .block_size = sizeof(SVC_DevRTCAlarmSetInput),
        .alarm = *alarm,
        .notify_proxy = notify_proxy,
        .notify_closure = notify_closure,
    } ;

    SVC_DevRequest rtc_req = dev_req_encode(DEV_RTC_CLASS, rtc_alarm_set, rtc) ;
    return dev_realm_svc_call(rtc_req, &input, NULL, NULL) ;
}
```

```
<<dev realm rtc proxy: static function definitions>>=
static int
dev_realm_rtc_alarm_set(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    RTC_ControlBlock *const rcb = &rtcs[inst] ;
    RTC_Type *const rtc = rcb->hdwr_controls.rtc_regs ;

    NVIC_DisableIRQ(rcb->hdwr_controls.irq_num) ;
    rtc->INTEN = 0 ;

    send_rtc_notification(rcb, alarm_discarded) ;
    rcb->notify_proxy = NULL ;

    SVC_DevRTCAlarmSetInput rtc_input ;
    int status = copy_in_svc_param(input, sizeof(rtc_input), &rtc_input) ;
    rtcheck_zero_return(status, status) ;

    rtcheck_return(rtc_input.notify_proxy != NULL, -ERR_INVALID_PARAM) ;

    rcb->notification.notify_proxy = (SVC_DevNotifyProxy)rtc_input.notify_proxy ;
    rcb->notification.notify_closure = rtc_input.notify_closure ;

    rtc->INTCLR = RTC_INTCLR_ALM_Msk ;
    status = rtc_update_alarm(rcb, &rtc_input.alarm) ;
    if (status == 0) {
        rtc->INTEN = RTC_INTEN_ALM_Msk ;
        NVIC_EnableIRQ(rcb->hdwr_controls.irq_num) ;
```

```
    }

    return status ;
}
```

Canceling an alarm

```
<<dev svc rtc req: external declarations>>=
extern int
dev_rtc_alarm_cancel(
    SVC_DevInstance rtc) ;
```

rtc
The instance number of the RTC peripheral for which the alarm is to be cancelled.

```
<<dev svc rtc req: external definitions>>=
int
dev_rtc_alarm_cancel(
    SVC_DevInstance rtc)
{
    SVC_DevRequest rtc_req = dev_req_encode(DEV_RTC_CLASS, rtc_alarm_cancel, rtc) ;
    return dev_realm_svc_call(rtc_req, NULL, NULL, NULL) ;
}
```

```
<<dev realm rtc proxy: static function definitions>>=
static int
dev_realm_rtc_alarm_cancel(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(input == NULL) ; (void)input ;
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    RTC_ControlBlock *const rcb = &rtcs[inst] ;
    RTC_Type *const rtc = rcb->hdwr_controls.rtc_regs ;

    NVIC_DisableIRQ(rcb->hdwr_controls.irq_num) ;
    rtc->INTEN = 0 ;

    uint32_t rtcctl = rtc->RTCCTL ;
    rtcctl = BTWD_FIELD_INSERT(rtcctl, alarm_disabled, RTC_RTCCTL_RPT) ;
    rtc->RTCCTL = rtcctl ;

    send_rtc_notification(rcb, alarm_discarded) ;
    rcb->notification.notify_proxy = NULL ;

    return 0 ;
}
```

Dispatching RTC Requests

In keeping with the established pattern for device operations, RTC requests have an additional level of dispatch and a function is required to find the foreground proxy function associated with the device operation.

```
<<dev realm rtc proxy: external declarations>>=
extern int
dev_realm_rtc(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

The implementation of the RTC device realm dispatch function follows the same pattern used for other devices. It is a simple jump table held in an array.

```
<<dev realm rtc proxy: external function definitions>>=
int
dev_realm_rtc(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    int status = dev_req_validate(req, rtc_operation_count, RTC_INSTANCES) ;
    rtcheck_zero_return(status, status) ;

    static const SVC_DevRequestProxy rtc_proxies[rtc_operation_count] = {
        [rtc_set] = dev_realm_rtc_set,
        [rtc_get] = dev_realm_rtc_get,
        [rtc_alarm_set] = dev_realm_rtc_alarm_set,
        [rtc_alarm_cancel] = dev_realm_rtc_alarm_cancel,
    } ;

    SVC_DevOperation rtc_operation = dev_req_extract_operation(req) ;
    return rtc_proxies[rtc_operation](req, input, output, error) ;
}
```

The RTC device is given a class identifier.

```
<<dev realm param: constants>>=
#define DEV_RTC_CLASS        2
```

And the pointer to its dispatch function is inserted into the device request dispatch jump table.

```
<<svc entry: device request classes>>=
[DEV_RTC_CLASS] = dev_realm_rtc,
```

RTC IRQ Handling

IRQ handling for the RTC only involves clearing the interrupt and sending a background notification.

```
<<dev realm rtc proxy: external function definitions>>=
void
RTC_IRQHandler(void)
{
    RTC_ControlBlock *const rcb = &rtcs[0] ;                // ❶
    RTC_Type *const rtc = rcb->hdwr_controls.rtc_regs ;

    rtc->INTCLR = RTC_INTCLR_ALM_Msk ;

    assert(rcb->notification.notify_proxy != NULL) ;
    if (rcb->notification.notify_proxy != NULL) {
        send_rtc_notification(rcb, alarm_expired) ;
    } else {
```

```
        NVIC_DisableIRQ(rcb->hdwr_controls.irq_num) ;
        BTWD_CLEAR_REG_FIELD(&rtc->INTEN, RTC_INTEN_ALM) ;
    }

}
```

❶ There is, after all, only one RTC peripheral in the Apollo 3 SOC.

Code Layout

The code layout follows the pattern for device control code. Five files are created to handle the background requests, foreground proxies, and the interface information shared between the background and foreground code.

RTC service requests

```
<<dev_svc_rtc_req.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Function prototypes for RTC service requests.
 *--
 */
#ifndef DEV_SVC_RTC_REQ_H_
#define DEV_SVC_RTC_REQ_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "dev_svc_req.h"
#include "dev_realm_rtc_param.h"
/*
 * Data Type Declarations
 */
<<dev svc rtc req: data type declarations>>
/*
 * External Declarations
 */
<<dev svc rtc req: external declarations>>

#endif /* DEV_SVC_RTC_REQ_H_ */
```

RTC Device Service Requests

```
<<dev_svc_rtc_req.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
```

```
 * Project:
 *   Code to Models
 *
 * Module:
 *   Device Realm RTC Request Implementation
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "useful.h"
#include "svc_req_errors.h"
#include "svc_req.h"
#include "dev_svc_req.h"
#include "dev_svc_rtc_req.h"
#include "dev_realm_rtc_param.h"
/*
 * External Functions
 */
<<dev svc rtc req: external definitions>>
```

Device Realm RTC Parameters

```
<<dev_realm_rtc_param.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Parameter interface data structures for RTC device requests.
 *--
 */
#ifndef DEV_REALM_RTC_PARAM_H_
#define DEV_REALM_RTC_PARAM_H_
/*
 * Include Files
 */
#include "dev_realm_param.h"
/*
 * Constants
 */
<<dev realm rtc param: constants>>
/*
 * Data Type Declarations
 */
<<dev realm rtc param: data type declarations>>

#endif /* DEV_REALM_RTC_PARAM_H_ */
```

Device Realm RTC Proxy Header

```
<<dev_realm_rtc_proxy.h>>=
<<edit warning>>
```

```
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Interfaces for device realm rtc proxy functions.
 *--
 */
#ifndef DEV_REALM_RTC_PROXY_H_
#define DEV_REALM_RTC_PROXY_H_

/*
 * Include files
 */
#include "dev_realm_param.h"
/*
 * External Declarations
 */
<<dev realm rtc proxy: external declarations>>

#endif /* DEV_REALM_RTC_PROXY_H_ */
```

```
<<svc handler: device include files>>=
#include "dev_realm_rtc_proxy.h"
```

Device Realm RTC Proxy Code Layout

```
<<dev_realm_rtc_proxy.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for device realm RTC proxy functions.
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "svc_req_errors.h"
#include "svc_proxy.h"
#include "dev_realm_proxy.h"
#include "dev_realm_rtc_param.h"
#include "bg_req_queue.h"
#include "panic.h"
#include "apollo3.h"
#include "bit_twiddle.h"
#include "sys_twiddle.h"
#include "dev_twiddle.h"
#include "useful.h"
#include "rtcheck.h"
/*
 * Constants
```

```
 */
<<dev realm rtc proxy: constants>>
/*
 * Data Type Declarations
 */
<<dev realm rtc proxy: data type declarations>>
/*
 * Forward References
 */
<<dev realm rtc proxy: forward references>>
/*
 * Static Data Definitions
 */
<<dev realm rtc proxy: static data definitions>>
/*
 * Static Inline Function Definitions
 */
<<dev realm rtc proxy: static inline functions>>
/*
 * Static Function Definitions
 */
<<dev realm rtc proxy: static function definitions>>
/*
 * External Function Definitions
 */
<<dev realm rtc proxy: external function definitions>>
```

Testing

To test the RTC functions, the Unity testing framework is used again.

```
<<rtc-test: static functions>>=
void
setUp(void)
{
}
```

```
<<rtc-test: static functions>>=
void
tearDown(void)
{
}
```

Test cases

```
<<rtc-test: static data>>=
static RTC_Time const current = {
    .year = 2021,
    .month = 12,
    .date = 31,
    .hour = 23,
    .minute = 59,
    .second = 59,
    .hundredth = 0,
    .weekday = 2,
} ;
```

```
<<rtc-test: static functions>>=
static void
set_rtc_time(void) {
    int status = dev_rtc_set(0, &current) ;
    TEST_ASSERT(status == 0) ;
}
```

```
<<rtc-test: static functions>>=
static void
get_rtc_time(void) {
    RTC_Time now ;

    int status = dev_rtc_get(0, &now) ;
    TEST_ASSERT(status == 0) ;

    TEST_ASSERT(current.year == now.year) ;
    TEST_ASSERT(current.month == now.month) ;
    TEST_ASSERT(current.date == now.date) ;
    TEST_ASSERT(current.hour == now.hour) ;
    TEST_ASSERT(current.minute == now.minute) ;
    TEST_ASSERT(current.second == now.second) ;
    TEST_ASSERT(current.weekday == now.weekday) ;

    struct tm time ;
    time.tm_sec = now.second ;
    time.tm_min = now.minute ;
    time.tm_hour = now.hour ;
    time.tm_mday = now.date ;
    time.tm_mon = now.month - 1 ;                          // ❶
    time.tm_year = now.year - 1900 ;
    time.tm_wday = now.weekday ;
    time.tm_yday = 0 ;
    time.tm_isdst = 0 ;

    char tbuf[32] ;
    (void)strftime(tbuf, sizeof(tbuf), "time: %F %T", &time) ;
    printf("%s\n", tbuf) ;
}
```

❶ *N.B.* the month and year conventions used by the POSIX `struct tm` time.

```
<<rtc-test: static data>>=
static bool volatile alarm_done ;
```

```
<<rtc-test: static functions>>=
static void
rtc_alarm_expired(
    SVC_DevRTCNotification const *const notify)
{
    bool volatile *done_ref = (bool volatile *)notify->notify_closure ;
    *done_ref = true ;

    struct tm time ;
    time.tm_sec = notify->time.second ;
    time.tm_min = notify->time.minute ;
    time.tm_hour = notify->time.hour ;
    time.tm_mday = notify->time.date ;
    time.tm_mon = notify->time.month - 1 ;
    time.tm_year = notify->time.year - 1900 ;
```

```
    time.tm_wday = notify->time.weekday ;
    time.tm_yday = 0 ;
    time.tm_isdst = 0 ;

    char tbuf[32] ;
    (void)strftime(tbuf, sizeof(tbuf), "%F %T", &time) ;

    switch (notify->status) {
    case alarm_expired:
        printf("alarm expired: %s\n", tbuf) ;
        break ;

    case alarm_discarded:
        printf("alarm discarded: %s\n", tbuf) ;
        break ;
    }
}
```

A series of one second RTC alarms

```
<<rtc-test: static functions>>=
static void
set_rtc_alarm(void) {
    RTC_Alarm alarm = {
        .repeat = alarm_every_second,
        .month = 0,
        .date = 1,
        .hour = 0,
        .minute = 0,
        .second = 0,
        .hundredth = 0,
        .weekday = 0,
    } ;

    int status = dev_rtc_alarm_set(0, &alarm, rtc_alarm_expired,
            (SVC_DevNotifyClosure)&alarm_done) ;
    TEST_ASSERT(status == 0) ;

    for (unsigned i = 0 ; i < 10 ; i++) {
        sys_ctrl_busy_wait(&alarm_done) ;
        TEST_ASSERT(alarm_done) ;
    }
}
```

Cancel RTC alarm

```
<<rtc-test: static functions>>=
static void
cancel_rtc_alarm(void) {
    int status = dev_rtc_alarm_cancel(0) ;
    TEST_ASSERT(status == 0) ;

    sys_ctrl_busy_wait(&alarm_done) ;
    TEST_ASSERT(alarm_done) ;
}
```

```
<<rtc-test: external functions>>=
int
main(
    int argc,
    char **argv)
{
```

```
#       ifdef USE_SEGGER_RTT
    SEGGER_RTT_Init() ;
#       endif /* USE_SEGGER_RTT */

    printf("beginning RTC tests run\n") ;

    UNITY_BEGIN() ;

    RUN_TEST(set_rtc_time) ;
    RUN_TEST(get_rtc_time) ;
    RUN_TEST(set_rtc_alarm) ;
    RUN_TEST(cancel_rtc_alarm) ;

    return UNITY_END() ;
}
```

RTC Tests Code Layout

```
<<time-mancery-rtc-test.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Tests for real time clock (RTC) device code.
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
#include <assert.h>
#include <stdio.h>
#include <inttypes.h>
#include <stdbool.h>
#include <time.h>
#include "sys_svc_req.h"
#include "dev_svc_rtc_req.h"
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
#include "unity.h"
/*
 * Constants
 */
<<rtc-test: constants>>
/*
 * Data Type Declarations
 */
<<rtc-test: data type declarations>>
/*
 * External Declarations
 */
<<rtc-test: external declarations>>
/*
 * Forward References
```

```
 */
<<test util: forward references>>
<<rtc-test: forward references>>
/*
 * Static Inline Functions
 */
<<rtc-test: static inline functions>>
/*
 * External Inline Functions
 */
<<rtc-test: external inline functions>>
/*
 * Static Data
 */
<<rtc-test: static data>>
/*
 * Static Functions
 */
<<test util: static functions>>
<<rtc-test: static functions>>
/*
 * External Functions
 */
<<rtc-test: external functions>>
```

Computer Time

The first microprocessors were simple enough devices that instruction execution timing was determined solely by the frequency of the system clock. With careful, albeit tedious, coding, instruction execution could be used as a means to measure time. Modern devices, with caches, wait states, and performance enhancing features, no longer have a direct relationship between instruction execution and system clock cycles. Counting instructions makes for a poor approximation to time. In addition to being wasteful of power, timer peripherals just do a better job.

However, the performance of software still has a direct relationship to the number of system clock cycles required to execute a given piece of code. Knowledge of that relationship is required to make informed engineering trade-offs between speed and memory. For the ARM v7-M architecture, there is a cycle counter which counts the number of system clock ticks as a program executes. With knowledge of the system clock frequency, the cycle counter is an accurate representation of execution time.

Our primary tool for measuring execution is a built-in cycle counter. In this chapter, a set of system realm request function to access the cycle counter which resides in the Data Watchpoint and Trace (DWT) unit of the Cortex-M4 processor are provided. There is one difficulty that you notice immediately. Access to the cycle counter is a privileged operation. This means foreground requests via the `SVC` exception must be made to operate on the cycle counter. This implies that the number of cycles measured is also going to contain cycles associated with leaving privileged context after the foreground request has manipulated the counter. This implies that only relative comparisons ,rather than the absolute numbers, of cycle counts are meaningful since the cycle counts includes the overhead of the foreground request mechanisms.

Enabling the cycle counter

```
<<sys svc req: external declarations>>=
extern int
sys_cyccnt_control(
    bool enable) ;
```

enable
A boolean value which indicates whether the cycle counter is to be enable (`true`) or disabled (`false`).

The `sys_cyccnt_enable` function enables or disables the internal processor clock cycle counter. The function returns zero upon success and a negative value otherwise.

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysCyccntControlInput,
    bool enable ;
) ;
```

```
<<sys svc req: external definitions>>=
int
sys_cyccnt_control(
    bool enable)
{
    SVC_SysCyccntControlInput input = {
        .block_size = sizeof(SVC_SysCyccntControlInput),
        .enable = enable,
    } ;

    return sys_realm_svc_call(SYS_CYCCNT_CONTROL, &input, NULL, NULL) ;
}
```

Since this is a system realm request, it is assigned a unique number and must be entered into the jump table of request functions.

```
<<sys realm param: constants>>=
#define SYS_CYCCNT_CONTROL   7
```

```
<<svc entry: system request functions>>=
[SYS_CYCCNT_CONTROL] = sys_realm_cyccnt_control,
```

The foreground proxy function has the usual interface.

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_cyccnt_control(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_cyccnt_control(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(!BTWD_TEST_REG_FIELD(&DWT->CTRL, DWT_CTRL_NOCYCCNT)) ;

    (void)req ;
    assert(output == NULL) ; (void)output ;
```

```
    assert(error == NULL) ; (void)error ;

    SVC_SysCyccntControlInput ctrl_input ;
    int status = copy_in_svc_param(input, sizeof(ctrl_input), &ctrl_input) ;
    rtcheck_zero_return(status, status) ;

    if (ctrl_input.enable) {
        BTWD_SET_REG_FIELD(&CoreDebug->DEMCR, CoreDebug_DEMCR_TRCENA) ;
        BTWD_SET_REG_FIELD(&DWT->CTRL, DWT_CTRL_CYCCNTENA) ;
        DWT->CYCCNT = 0 ;
    } else {
        BTWD_CLEAR_REG_FIELD(&CoreDebug->DEMCR, CoreDebug_DEMCR_TRCENA) ;
        BTWD_CLEAR_REG_FIELD(&DWT->CTRL, DWT_CTRL_CYCCNTENA) ;
    }

    return 0 ;
}
```

Reading the cycle counter

```
<<sys svc req: external declarations>>=
extern int
sys_cyccnt_read(
    uint32_t *const counter_ref,
    bool zero) ;
```

counter_ref
: A pointer to a memory object where the cycle count is placed after it is read. If `counter_ref` is `NULL`, then no counter value is returned and the action of the function is solely dependent upon the `zero` setting.

zero
: A boolean value which indicates whether the cycle counter is to be set to zero after it is read.

The `sys_cyccnt_read` function reads the current value of the cycle counter register. After reading, the cycle counter is set to zero if the `zero` argument is `true`. If `counter_ref` is not `NULL` then the value of the cycle counter is placed in the object pointed to by `counter_ref`.

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysCyccntReadInput,
    bool zero ;
) ;
```

```
<<sys realm param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_SysCyccntReadOutput,
    uint32_t counter ;
) ;
```

```
<<sys svc req: external definitions>>=
int
sys_cyccnt_read(
    uint32_t *const counter_ref,
    bool zero)
{
    SVC_SysCyccntReadInput input = {
        .block_size = sizeof(SVC_SysCyccntReadInput),
        .zero = zero,
```

```
    } ;

    SVC_SysCyccntReadOutput output = {
        .block_size = sizeof(SVC_SysCyccntReadOutput),
        .counter = 0,
    } ;

    return sys_realm_svc_call(SYS_CYCCNT_READ, &input, &output, NULL) ;
}
```

Since this is a system realm request, it is assigned a unique number and must be entered into the jump table of request functions.

```
<<sys realm param: constants>>=
#define SYS_CYCCNT_READ   8
```

```
<<svc entry: system request functions>>=
[SYS_CYCCNT_READ] = sys_realm_cyccnt_read,
```

The foreground proxy function has the usual interface.

```
<<sys realm proxy: external declarations>>=
extern int
sys_realm_cyccnt_read(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<sys realm proxy: external function definitions>>=
int
sys_realm_cyccnt_read(
    SVC_SysRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    (void)req ;
    assert(output == NULL) ; (void)output ;

    SVC_SysCyccntReadInput read_input ;
    int status = copy_in_svc_param(input, sizeof(read_input), &read_input) ;
    rtcheck_zero_return(status, status) ;

    SVC_SysCyccntReadOutput read_output = {
        .block_size = sizeof(SVC_SysCyccntReadOutput),
        .counter = DWT->CYCCNT,
    } ;

    status = copy_out_svc_result(output, sizeof(read_output), &read_output) ;

    if (read_input.zero) {
        DWT->CYCCNT = 0 ;
    }

    return status ;
}
```

Summary

Summary

Chapter 8

The Tyranny of the Pins

In the late 1950's and into the 1960's CE, computer designers were struggling against the shear number of wires which had to be connected in order to build a computer from the electronic components available at the time.

> For some time now, electronic man has known how *in principle* to extend greatly his visual, tactile, and mental abilities through the digital transmission and processing of all kinds of information. However, all these functions suffer from what has been called *the tyranny of numbers.* Such systems, because of their complex digital nature, require hundreds, thousands, and sometimes tens of thousands of electron devices.
>
> — Jack Morton *The Tyranny of Numbers -- 1958*

This time was before the invention of integrated circuits, which went a long way toward solving these problems. However, computer chips, especially for systems which interact directly with the outside environment, must be electrically wired to that environment. That is usual done by connecting a *pin* from the chip to some external circuitry. There are only so many pins which can be reasonably accommodated on the physical chip carrier. Physical size is always a constraint in an overall system design and genuine electrical and mechanical constraints apply. The external chip pins must be connected to the internal chip peripheral which uses them. The total number of pins required if all a chip's peripherals were in use usually exceeds the number of pins on the physical package. Chips are designed to be used in a wide range of systems and most chip designs depend upon systems which do not need every peripheral designed into an SOC. Some form of multiplexing of the physical I/O pins to the peripheral I/O signals is a common solution. The mechanisms and interfaces to select the mapping between the physical I/O pin and its internal function vary significantly between chip vendors.

It would be convenient if I/O pins could be treated the same. Unfortunately, the electrical characteristics required don't allow that. The use for a pin must be specified as input or output. There are usually controls over how much current an output pin can drive. Inevitably there are *special* rules about certain pin usage and analog I/O also imposes different conditions.

To compound the problem, when a chip is used on a particular board, the I/O connections are usually given names which indicate their use on the board. So, usually there is a name mapping which must be considered. Tracing through the variety of names a pin might have in different contexts is an annoyingly necessary undertaking.

For our case, the Apollo 3 has 50 I/O pins. Each I/O pin has up to 7 different selectable functions. In addition, when the Apollo 3 is packaged as the SparkFun Artemis module and used as a MicroMod processor, only certain pins are available and they are mapped to MicroMod naming conventions.

Much of the details of this part of the book is necessarily specific to the Apollo 3 and the manner in which it controls I/O functionality. Consider the built-in LED on the Artemis processor board as an example. The following figure is from the processor board schematic diagram.

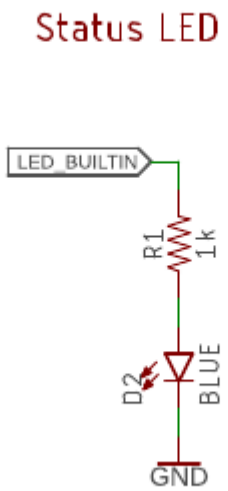

Figure 8.1: Schematic Diagram for Micromod Status LED

The symbology of schematic diagrams is not discussed here. The information of interest shows that a pin named, `LED_BUILTIN`, is connected to an LED which is mounted on the processor board. To control the LED[1], what LED_BUILTIN is connected to must be found.

The following figure shows the Artemis module.

[1] From the schematic it appears that setting the `LED_BUILTIN` pin to 1 causes a blue LED to illuminate.

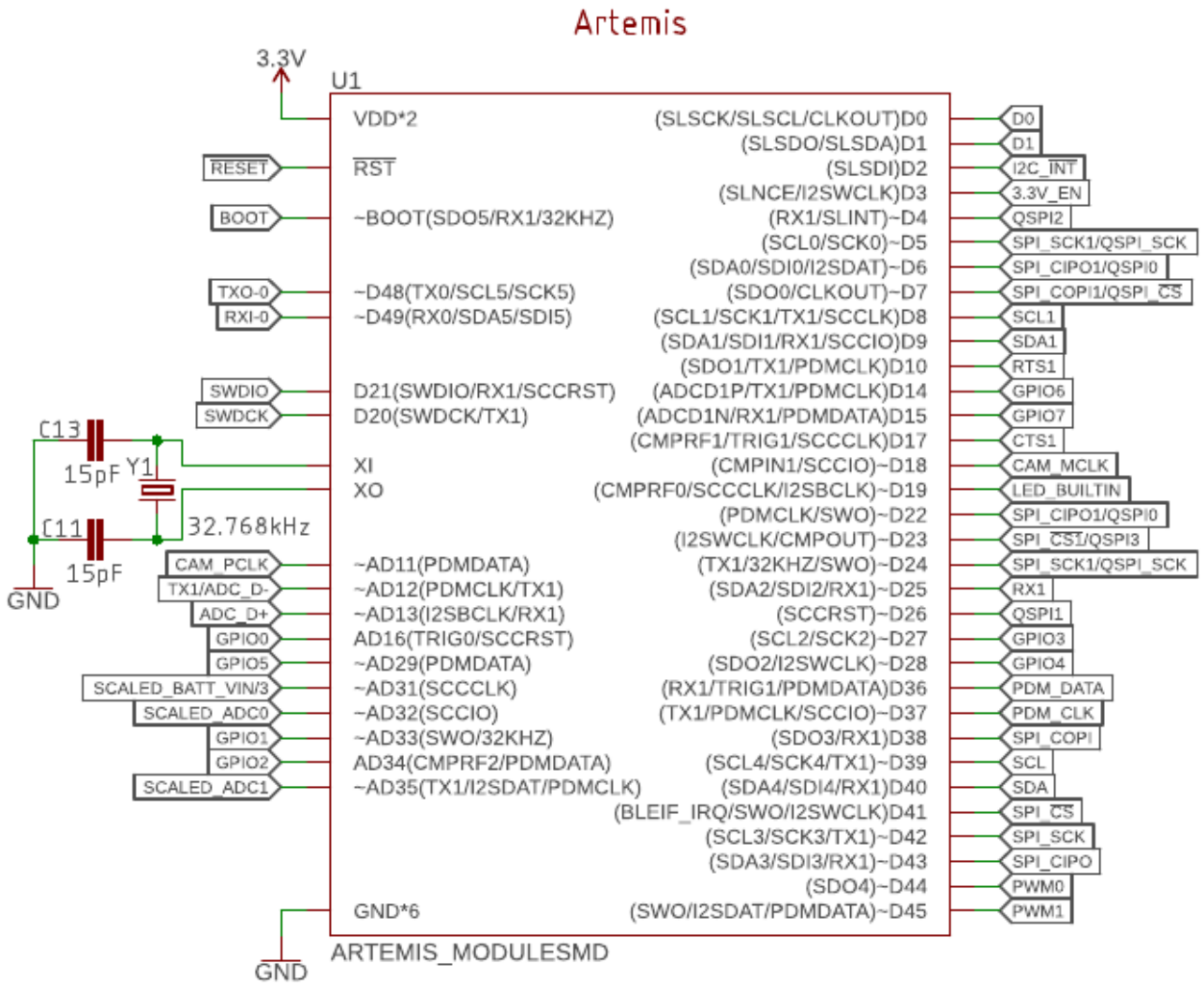

Figure 8.2: Schematic Diagram for Micromod Artemis Module

The annotations on the inside of the module *box* show the pin labeling from the point of view of the Apollo 3 processor. The tags outside the module box show the labeling from the point of view of the processor board schematic. Careful examination shows on the right hand side of the module box that `LED_BUILTIN` connects to the `D19(CMPRFO/SCCLK/I2SBCLK).D19` is the pin designation. `CMPRFO/SCCLK/I2SBCLK` are abbreviations for some of the internal functions for which `D19` may be used (usually each pin has 7 possible functions). So, to control the on-board LED, pin `D19` must be configured as a general purpose output.

This chapter shows a set of background request functions which form the interface for controlling processor pins for General Purpose Input/Output (GPIO) functionality. Note that no attempt is made to provide a means to configure all possible functions of the GPIO pins. The Apollo 3 GPIO pins have many different and specialized configurations. The many pins have specialized uses. The complexity of the general case is somewhat dizzying. Rather, configuring and using the processor pins as simple I/O pins is sufficient. Later, when other peripheral devices are included, those request functions will handle the required configuration of I/O pins specific to the peripheral function. Some generic pin functions are provided which are useful for both this general purpose I/O pin usage and pin usage by other peripherals.

CMSIS definitions

One of the complexities of the Apollo 3 pin function selection scheme is that each pin has a distinct mapping of a 3-bit number to its set of uses. If the pin is used as an external I/O connection, then its function control is encoded as 3. This applies to all the pins. But for any internal peripheral use, the encoding of the function is not the same across all pins and a detailed mapping is required. For example, pin 19, can be used as the receive pin for UART #1 if its function selector is set to 6. Setting the function selector for pin 18 to 6 connects the pin to the transmit pin for UART #2. The point here is that you must have the mapping table from in the Apollo 3 data sheet to determine the function selector value that allows the pin to be used with a particular peripheral.

This variation of the function encoding from pin to pin complicates the CMSIS description. The Ambiq solution to this problem is to encode the registers which contain the pin function selection to uniquely describe the allowed values for that pin. To add to the complexity, the memory mapped hardware interface groups multiple pin controls together and packs them into a single register. The Ambiq SDK contains HAL code to configure an arbitrary pin to any allowed setting. It is a complex undertaking to enforce all the special rules for how the Apollo 3 pins work. The GPIO register layout in the Ambiq-supplied SVC file makes it very difficult to write the usual generic bit manipulation code using CMSIS pre-processor symbols without making assumptions about the organization of the I/O registers used to configure a pin.

In this design, the complexity is side-stepped by changing the Ambiq Apollo 3 SVD file to treat the configuration registers as an array and handle the differences in pin function encoding and other special cases explicitly in code. The changed SVD file is then used by the `SVDConv` program to generate the alternative CMSIS header files. Details of SVD files and their use in generating CMSIS device files can be found in the CMSIS-SVD documentation.

GPIO Operations

There are a number of GPIO register operations which can be factored into common code for use here and for the configuration needs of peripheral devices.

Allocating Pins

The Apollo 3 has 50 GPIO pins.

```
<<dev realm gpio param: constants>>=
#define GPIO_PIN_INSTANCES   50U
```

Which pins are in use is one global piece of needed information. Foreground requests for the various devices are expected to allocate and free pins as they are needed and to check the allocation they want to use.

The allocation information can be stored in a 64-bit integer used as a bit vector.

```
<<dev realm gpio proxy: static data definitions>>=
static uint64_t gpio_pin_allocations =
    (UINT64_C(1) << 20) |                  // ❶
    (UINT64_C(1) << 21) ;
```

❶ Pads 20 and 21 are connected the software debug clock and data, respectively at reset time. They are *not* to be reallocated except with considerable deliberation.

```
<<dev realm gpio proxy: external declarations>>=
extern int
gpio_pin_alloc(
    unsigned pin) ;
```

pin
A pin number (0 - 49) designating which GPIO pin to mark as allocated.

The `gpio_pin_alloc` function marks the GPIO pin given by, `pin`, as allocated. The function returns 0 upon success and a negative value if the pin is already allocated.

```
<<dev realm gpio proxy: external function definitions>>=
int
gpio_pin_alloc(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint64_t mask = UINT64_C(1) << pin ;
    if ((gpio_pin_allocations & mask) != 0) {
        return -ERR_OPERATION_FAILED ;
    }

    gpio_pin_allocations |= mask ;
    return 0 ;
}
```

```
<<dev realm gpio proxy: external declarations>>=
extern void
gpio_pin_free(
    unsigned pin) ;
```

pin
A pin number (0 - 49) designating which GPIO pin is to be de-allocated.

The `gpio_pin_free` function marks the GPIO pin given by, `pin`, as free so that it may be reallocated in the future.

```
<<dev realm gpio proxy: external function definitions>>=
void
gpio_pin_free(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint64_t mask = UINT64_C(1) << pin ;
    gpio_pin_allocations &= ~mask ;
}
```

```
<<dev realm gpio proxy: external declarations>>=
extern bool
gpio_pin_is_allocated(
    unsigned pin) ;
```

pin
A pin number (0 - 49) designating which GPIO pin is to be checked for allocation.

The `gpio_pin_is_allocated` function returns `true` if `pin` is allocated and `false` otherwise.

```
<<dev realm gpio proxy: external function definitions>>=
bool
gpio_pin_is_allocated(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint64_t mask = UINT64_C(1) << pin ;
    return (gpio_pin_allocations & mask) != 0 ;
}
```

GPIO pin to register field mapping

Pin configuration for the Apollo 3 is controlled primarily with three registers, `PADREG`, `CFG`, and `ALTPADCFG`. Note that there is a difference between configuring the *pad* of the GPIO and configuring the *pin*. The reconfigured SVD file groups these registers as arrays since they are located sequentially in the memory. There are 13 `PADREG` and `ALTPADCFG` instances and 7 `CFG` instances. Each `PADREG` register is further divided into 8-bit groups which control the settings for one pin. The 8-bit groups are further divided into 4 bit fields which apply to all pins and a fifth field is used to control functions which are particular to only certain pins. The `ALTPADCFG` registers are also divided into 8-bit groups with additional bit fields in the group, but the controls are the same for all pins. The `CFG` registers are divided into 4-bit groups with additional control bit fields in each group. Some of the control bits in the `CFG` groups are overloaded in their control meaning. Yes, it is a complicated interfacing scheme.

The functions in this section are used to compute a pointer to the specific instance of one of the control registers and an offset to where the group of control bits is located within the register. The functions compute, for a given pin number, a register address and an offset within that register where the control bit field is located. The implementation is devolves to quotient and modulus arithmetic with defined constants for the internal register groupings since the modified CMSIS definitions structure the registers sets as arrays.

```
<<dev realm gpio param: static inline functions>>=
static inline uint32_t volatile *
gpio_pad_reg(
    unsigned pin,
    unsigned *offset)
{
    assert(pin < GPIO_PIN_INSTANCES) ;
    assert(offset != NULL) ;

    *offset = (pin % PADREG_GROUP_COUNT) * PADREG_GROUP_WIDTH ;
    return &GPIO->PADREG[pin / PADREG_GROUP_COUNT] ;
}
```

pin
A pin number (0 - 49) designating a GPIO pin.

offset
A pointer to an unsigned object where the offset to the control group for `pin` is returned.

The `gpio_pad_reg` computes the register address and bit offset to the `PADREG` control group for the GPIO pin given by, `pin`. The control group offset within the register is returned by reference using the `offset` pointer.

```
<<dev realm gpio param: static inline functions>>=
static inline uint32_t volatile *
gpio_alt_pad_reg(
    unsigned pin,
    unsigned *offset)
{
    assert(pin < GPIO_PIN_INSTANCES) ;
    assert(offset != NULL) ;

    *offset = (pin % ALTPADREG_GROUP_COUNT) * ALTPADREG_GROUP_WIDTH ;
    return &GPIO->ALTPADCFG[pin / ALTPADREG_GROUP_COUNT] ;
}
```

pin
A pin number (0 - 49) designating a GPIO pin.

offset
A pointer to an unsigned object where the offset to the control group for `pin` is returned.

The `gpio_alt_pad_reg` computes the register address and bit offset to the `ALTPADCFG` control group for the GPIO pin given by, `pin`. The control group offset within the register is returned by reference using the `offset` pointer.

```
<<dev realm gpio param: static inline functions>>=
static inline uint32_t volatile *
gpio_cfg_reg(
    unsigned pin,
    unsigned *offset)
{
    assert(pin < GPIO_PIN_INSTANCES) ;
    assert(offset != NULL) ;

    *offset = (pin % CFG_GROUP_COUNT) * CFG_GROUP_WIDTH ;
    return &GPIO->CFG[pin / CFG_GROUP_COUNT] ;
}
```

pin
A pin number (0 - 49) designating a GPIO pin.

offset
A pointer to an unsigned object where the offset to the control group for `pin` is returned.

The `gpio_cfg_reg` computes the register address and bit offset to the `CFG` control group for the GPIO pin given by, `pin`. The control group offset within the register is returned by reference using the `offset` pointer.

Bit definitions for the offsets and widths of the control fields within a group are also needed. Where possible, the CMSIS information for the 0^{th} field is used.

```
<<dev realm gpio param: constants>>=
#define  PADREG_FNCSEL_GPIO     3U

#define PADREG_PULL_OFFSET      GPIO_PADREG_PAD0PULL_Pos
#define PADREG_INPEN_OFFSET     GPIO_PADREG_PAD0INPEN_Pos
#define PADREG_STRNG_OFFSET     GPIO_PADREG_PAD0STRNG_Pos
#define PADREG_FNCSEL_OFFSET    GPIO_PADREG_PAD0FNCSEL_Pos
#define PADREG_FNCSEL_WIDTH     3U
#define PADREG_RSEL_OFFSET      GPIO_PADREG_PAD0RSEL_Pos
#define PADREG_RSEL_WIDTH       2U
#define PADREG_GROUP_WIDTH      8U
#define PADREG_GROUP_COUNT      (32U / PADREG_GROUP_WIDTH)

#define CFG_INCFG_OFFSET        0
#define CFG_OUTCFG_OFFSET       1U
#define CFG_OUTCFG_WIDTH        2U
#define CFG_INTD_OFFSET         3U
#define CFG_GROUP_WIDTH         4U
#define CFG_GROUP_COUNT         (32U / CFG_GROUP_WIDTH)

#define ALTPADCFG_DS1_OFFSET    GPIO_ALTPADCFG_PAD0_DS1_Pos
#define ALTPADREG_GROUP_WIDTH   8U
#define ALTPADREG_GROUP_COUNT   (32U / ALTPADREG_GROUP_WIDTH)
```

GPIO pin operations

This section presents present a set of operations for a single GPIO pin. Since the state of a pin can be represented by a single bit, the operations usually result in familiar bit mask expressions. The Apollo 3 GPIO's also supply registers which can set or clear a pin while not affecting other pins. These registers save a read/modify/write transaction on the GPIO register. Where applicable, these functions take advantage of such specialized registers.

```
<<dev realm gpio proxy: static function definitions>>=
static void
```

```
gpio_pin_set(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    uint32_t volatile *const set_reg = &GPIO->WTS[index] ;

    *set_reg = mask ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static void
gpio_pin_clear(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    uint32_t volatile *const clear_reg = &GPIO->WTC[index] ;

    *clear_reg = mask ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static void
gpio_pin_toggle(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    uint32_t volatile *const write_reg = &GPIO->WT[index] ;

    *write_reg ^= mask ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static void
gpio_tristate_set(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    uint32_t volatile *const set_reg = &GPIO->ENS[index] ;

    *set_reg = mask ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static void
gpio_tristate_clear(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;
```

```
    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    uint32_t volatile *const clear_reg = &GPIO->ENC[index] ;

    *clear_reg = mask ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static void
gpio_tristate_toggle(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    uint32_t volatile *const write_reg = &GPIO->EN[index] ;

    *write_reg ^= mask ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static unsigned
gpio_pin_read(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    uint32_t volatile *const read_reg = &GPIO->RD[index] ;

    return (*read_reg & mask) == 0 ? 0 : 1U ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static void
gpio_pin_intr_disable(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    GPIO->INTCTRL[index].INTEN &= ~mask ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static void
gpio_pin_intr_enable(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    uint32_t mask = btwd_bit_mask(pin % 32U) ;
    uint32_t index = pin / 32U ;
    GPIO->INTCTRL[index].INTCLR = mask ;
    GPIO->INTCTRL[index].INTEN |= mask ;
    NVIC_EnableIRQ(GPIO_IRQn) ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static void
gpio_pin_disable(
    unsigned pin)
{
    assert(pin < GPIO_PIN_INSTANCES) ;

    gpio_pin_intr_disable(pin) ;

    unsigned offset ;
    uint32_t volatile *reg ;

    reg = gpio_pad_reg(pin, &offset) ;
    uint32_t pad_reg_value = *reg ;
    pad_reg_value = btwd_bits_insert(pad_reg_value,
            PADREG_FNCSEL_GPIO << PADREG_FNCSEL_OFFSET,
            PADREG_GROUP_WIDTH, offset) ;
    *reg = pad_reg_value ;

    reg = gpio_cfg_reg(pin, &offset) ;
    uint32_t cfg_reg_value = *reg ;
    cfg_reg_value = btwd_bits_insert(cfg_reg_value,
            0, CFG_GROUP_WIDTH, offset) ;
    *reg = cfg_reg_value ;

    reg = gpio_alt_pad_reg(pin, &offset) ;
    uint32_t alt_pad_reg_value = *reg ;
    alt_pad_reg_value = btwd_bits_insert(alt_pad_reg_value,
            0, ALTPADREG_GROUP_WIDTH, offset) ;
    *reg = alt_pad_reg_value ;
}
```

GPIO Background Requests

The following diagram shows the usage of peripherals for the GPIO device background requests.

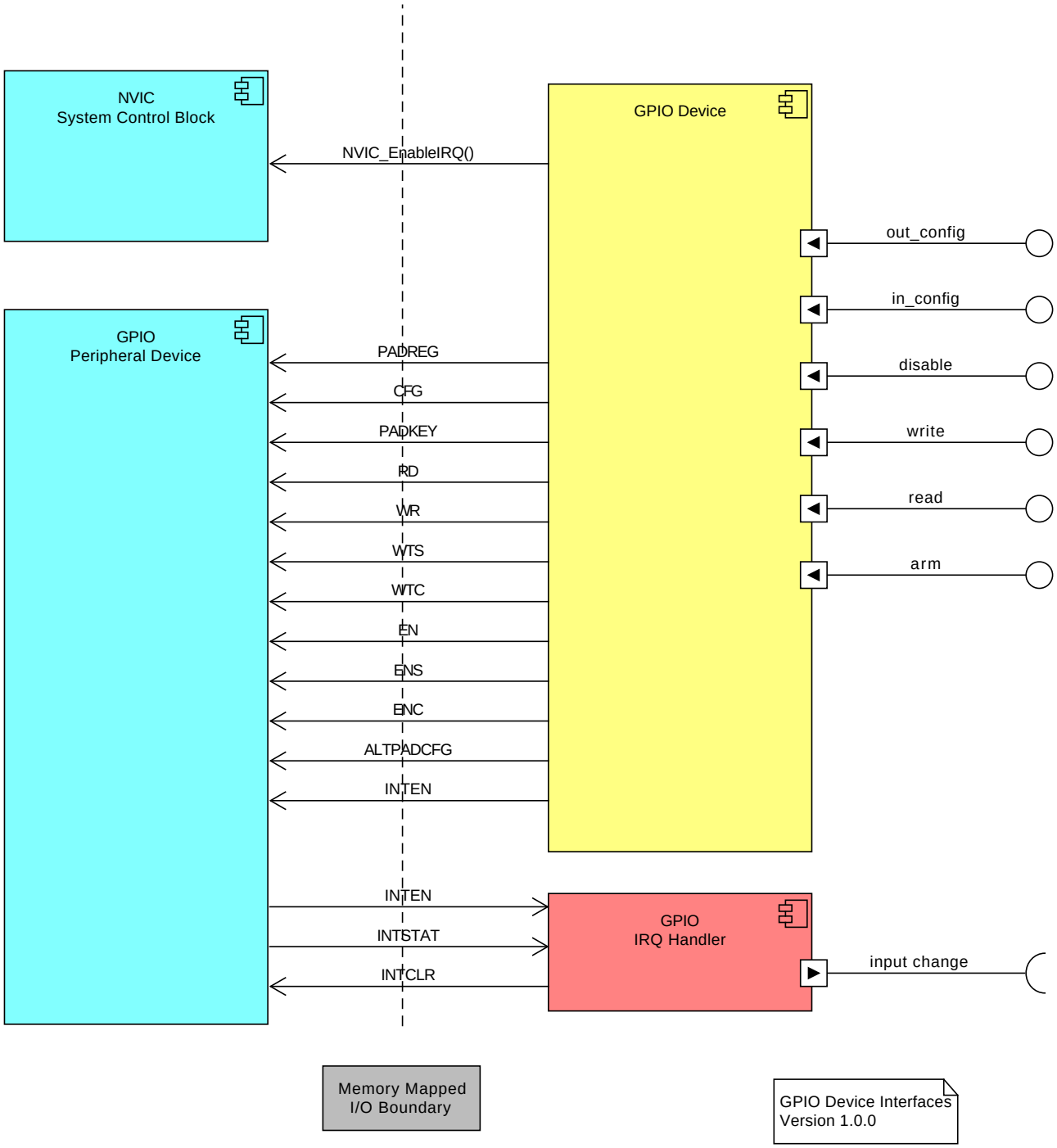

Figure 8.3: Snapshot of GPIO Interface Components

Notice that configuration request for output are separate from those for input. There is some overlap in configuration settings, but generally you know whether a pin is to be used for input or output from the context of the application. Separating the different configurations makes for a simpler interface.

As done previously for device realm requests, the operations are sequentially encoded using an enumerator.

```
<<dev realm gpio param: data type declarations>>=
```

```
enum SVC_gpioRequest {
    gpio_out_config = 0,
    gpio_in_config,
    gpio_disable,
    gpio_write,
    gpio_read,
    gpio_arm,

    gpio_operation_count        // last
} ;
```

The following sections present the background requests for the GPIO device. The familiar pattern of presenting the request function which executes unprivileged in the background followed by the foreground proxy function which manipulates the peripherals is followed.

Configure an Output Pin

```
<<dev svc gpio req: external declarations>>=
extern int
dev_gpio_out_config(
    SVC_DevInstance pin,
    GPOUT_OutputType type,
    PAD_DriveStrength drive_strength,
    PAD_PullupResistor pullup_resistor) ;
```

pin
: The GPIO pin number of the pin to be configured as an output.

type
: An indicator of the electrical characteristics of the pin output.

drive_strength
: An indication of the amount of current the pin is capable of driving.

pullup_resistor
: A value for any pull up resistor to be associated with the pin pad. *N.B.* for pin #20, the resistor is actually a pull-down resistor. For all other pins, the pad resistor serves as a pull-up resistor.

The `dev_gpio_out_config` function configures the GPIO `pin` to be an output pin. The manner in the output is driven is given by `type`. The amount of current which the pin can dirve is given by `drive_strength`. An pull-up resistor value for the GPIO pad is specified by `pullup_resistor`. The function returns zero on success and a negative error number otherwise.

Some encodings of the various arguments for the parameters to the function are required.

```
<<dev realm gpio param: data type declarations>>=
typedef enum {
    gpout_pushpull = 1,
    gpout_opendrain,
    gpout_tristate,
} GPOUT_OutputType ;
```

```
<<dev realm gpio param: data type declarations>>=
typedef enum {
    pad_pullup_none = 0,
    pad_pullup_weak,
    pad_pullup_1_5K,
```

```
    pad_pullup_6K,
    pad_pullup_12K,
    pad_pullup_24K,
} PAD_PullupResistor ;
```

```
<<dev realm gpio param: data type declarations>>=
typedef enum {
    pad_drive_2mA = 0,
    pad_drive_4mA,
    pad_drive_8mA,
    pad_drive_12mA,
} PAD_DriveStrength ;
```

Next, a data structure to pass the parameters across the SVC interface is needed.

```
<<dev realm gpio param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevGpioOutConfigInput,
    GPOUT_OutputType type ;
    PAD_DriveStrength drive_strength ;
    PAD_PullupResistor pullup_resistor ;
) ;
```

The background request function then marshals the parameters and passes them to the foreground proxy across the SVC call interface.

```
<<dev svc gpio req: external definitions>>=
int
dev_gpio_out_config(
    SVC_DevInstance pin,
    GPOUT_OutputType type,
    PAD_DriveStrength drive_strength,
    PAD_PullupResistor pullup_resistor)
{
    SVC_DevGpioOutConfigInput input = {
        .block_size = sizeof(SVC_DevGpioOutConfigInput),
        .type = type,
        .drive_strength = drive_strength,
        .pullup_resistor = pullup_resistor,
    } ;

    SVC_DevRequest gpio_req = dev_req_encode(DEV_GPIO_CLASS, gpio_out_config, pin) ;
    return dev_realm_svc_call(gpio_req, &input, NULL, NULL) ;
}
```

The Apollo 3 GPIO configuration is not orthogonal and many pins have special rules and uses. The first one encountered here is with respect to pull-up resistors on the GPIO pads. All pads have the ability to enable a pull-up resistor. For some pads, the pull-up is either disabled or, if enabled, is only a *weak* pull up. For pads which are can be used to drive I^2C or SPI output, there is a range of pull-up resistor values which can be set. A check whether the pull-up resistor value give in the configuration is suitable for a given pin is made. The following function uses a bit mask to encode those pins which have a range of pull-up resistor values.

```
<<dev realm gpio proxy: static function definitions>>=
static bool
gpio_pad_has_pullups(
    unsigned pin)
{
    static uint64_t const pins_with_opt_pullup =
        (UINT64_C(1) << 0) | (UINT64_C(1) << 1) | (UINT64_C(1) << 5) |
        (UINT64_C(1) << 6) | (UINT64_C(1) << 8) | (UINT64_C(1) << 9) |
        (UINT64_C(1) << 25) | (UINT64_C(1) << 27) | (UINT64_C(1) << 39) |
        (UINT64_C(1) << 40) | (UINT64_C(1) << 42) | (UINT64_C(1) << 43) |
```

```
        (UINT64_C(1) << 48) | (UINT64_C(1) << 49) ;

    uint64_t mask = UINT64_C(1) << pin ;
    return (pins_with_opt_pullup & mask) != 0 ;
}
```

A function to handle pad configuration for the pull-up resistors is factored out. It is used in two places.

```
<<dev realm gpio proxy: static function definitions>>=
static int
gpio_config_pullup_resistor(
    unsigned pin,
    PAD_PullupResistor pullup_resistor,
    uint32_t pad_cfg,
    uint32_t pad_offset,
    uint32_t *new_pad_cfg)
{
    assert(pin < GPIO_PIN_INSTANCES) ;
    assert(new_pad_cfg != NULL) ;

    unsigned pullup_offset = pad_offset + PADREG_PULL_OFFSET ;
    switch (pullup_resistor) {
    case pad_pullup_none:
        *new_pad_cfg = btwd_bit_clear(pad_cfg, pullup_offset) ;
        break ;

    case pad_pullup_weak:
        if (gpio_pad_has_pullups(pin)) {
            return -ERR_INVALID_PARAM ;
        }
        *new_pad_cfg = btwd_bit_set(pad_cfg, pullup_offset) ;
        break ;

    case pad_pullup_1_5K:       // fallthrough
    case pad_pullup_6K:         // fallthrough
    case pad_pullup_12K:        // fallthrough
    case pad_pullup_24K:
        if (!gpio_pad_has_pullups(pin)) {
            return -ERR_INVALID_PARAM ;
        }
        pad_cfg = btwd_bit_set(pad_cfg, pullup_offset) ;
        *new_pad_cfg = btwd_bits_insert(pad_cfg,
            (unsigned)pullup_resistor - (unsigned)pad_pullup_1_5K,      // ❶
            PADREG_RSEL_WIDTH, pad_offset + PADREG_RSEL_OFFSET) ;
        break ;

    default:
        panic("unknown pad pull up resistor value: %d\n", pullup_resistor) ;
        break ;
    }

    return 0 ;
}
```

❶ This construct is obscure and probably too clever for its own good. The encoding of the enumeration is transformed into the encoding needed by the hardware register. Changing the ordering of encoding values of the `PAD_PullupResistor` enumeration will *break* this expression.

```
<<dev realm gpio proxy: static function definitions>>=
static int
```

```
dev_realm_gpio_out_config(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevGpioOutConfigInput gpio_input ;
    int status = copy_in_svc_param(input, sizeof(gpio_input), &gpio_input) ;
    rtcheck_zero_return(status, status) ;

    SVC_DevInstance pin = dev_req_extract_instance(req) ;
    rtcheck_max_return(pin, GPIO_PIN_INSTANCES, -ERR_INVALID_PARAM) ;

    status = gpio_pin_alloc(pin) ;
    rtcheck_zero_return(status, status) ;

    unsigned pad_offset ;
    uint32_t volatile *const pad_cfg_reg = gpio_pad_reg(pin, &pad_offset) ;

    unsigned alt_pad_offset ;
    uint32_t volatile *const alt_pad_cfg_reg =
            gpio_alt_pad_reg(pin, &alt_pad_offset) ;

    unsigned cfg_offset ;
    uint32_t volatile *const cfg_reg = gpio_cfg_reg(pin, &cfg_offset) ;

    uint32_t pad_cfg = *pad_cfg_reg ;
    uint32_t alt_pad_cfg = *alt_pad_cfg_reg ;
    uint32_t cfg = *cfg_reg ;

    pad_cfg = btwd_bit_clear(pad_cfg, pad_offset + PADREG_INPEN_OFFSET) ; // ❶
    pad_cfg = btwd_bits_insert(pad_cfg, PADREG_FNCSEL_GPIO,
            PADREG_FNCSEL_WIDTH, pad_offset + PADREG_FNCSEL_OFFSET) ;
    pad_cfg = btwd_bits_insert(pad_cfg, 0,
            PADREG_RSEL_WIDTH, pad_offset + PADREG_RSEL_OFFSET) ;

    status = gpio_config_pullup_resistor(pin, gpio_input.pullup_resistor,
            pad_cfg, pad_offset, &pad_cfg) ;
    if (status != 0) {
        gpio_pin_free(pin) ;
        return status ;
    }

    unsigned pad_bit_offset = pad_offset + PADREG_STRNG_OFFSET ;
    unsigned alt_bit_offset = alt_pad_offset + ALTPADCFG_DS1_OFFSET ;
    switch (gpio_input.drive_strength) {
    case pad_drive_2mA:
        pad_cfg = btwd_bit_clear(pad_cfg, pad_bit_offset) ;
        alt_pad_cfg = btwd_bit_clear(alt_pad_cfg, alt_bit_offset) ;
        break ;

    case pad_drive_4mA:
        pad_cfg = btwd_bit_set(pad_cfg, pad_bit_offset) ;
        alt_pad_cfg = btwd_bit_clear(alt_pad_cfg, alt_bit_offset) ;
        break ;

    case pad_drive_8mA:
        pad_cfg = btwd_bit_clear(pad_cfg, pad_bit_offset) ;
        alt_pad_cfg = btwd_bit_set(alt_pad_cfg, alt_bit_offset) ;
```

```
        break ;

    case pad_drive_12mA:
        pad_cfg = btwd_bit_set(pad_cfg, pad_bit_offset) ;
        alt_pad_cfg = btwd_bit_set(alt_pad_cfg, alt_bit_offset) ;
        break ;

    default:
        return -ERR_INVALID_PARAM ;
    }

    cfg = btwd_bits_insert(cfg, gpio_input.type, CFG_OUTCFG_WIDTH,
            cfg_offset + CFG_OUTCFG_OFFSET) ;
    cfg = btwd_bit_set(cfg, cfg_offset + CFG_INCFG_OFFSET) ;
    cfg = btwd_bit_clear(cfg, cfg_offset + CFG_INTD_OFFSET) ;

    GPIO->PADKEY = GPIO_PADKEY_PADKEY_Key ;
    *pad_cfg_reg = pad_cfg ;
    *cfg_reg = cfg ;
    *alt_pad_cfg_reg = alt_pad_cfg ;
    GPIO->PADKEY = 0 ;

    return 0 ;
}
```

❶ When configuring for output, input is disabled.

Configure an Input Pin

To use a pin as an input, additional information must be supplied about how to inform the background when the input on the pin has changed state. First, the structure of the background notification is defined.

```
<<dev realm gpio param: data type declarations>>=
DECLARE_DEV_NOTIFICATION(SVC_DevGpioNotification,
    uint8_t pin_value ;
) ;
```

There is a corresponding function prototype which accepts the notification.

```
<<dev realm gpio param: data type declarations>>=
typedef void (*SVC_DevGpioNotifyProxy)(SVC_DevGpioNotification const *const params) ;
```

Interrupt can be generated on either transition, both transitions or no interrupt need be generated if the pin is to be polled only. Note if no interrupt is requested, then the proxy function arguments below may be `NULL`.

```
<<dev realm gpio param: data type declarations>>=
typedef enum {
    gpin_intr_none = 0,
    gpin_intr_both,
    gpin_intr_low_high,
    gpin_intr_high_low,
} GPIN_IntrConfig ;
```

```
<<dev svc gpio req: external declarations>>=
extern int
dev_gpio_in_config(
    SVC_DevInstance pin,
    GPIN_IntrConfig intr_config,
    bool disarm_on_active,
    PAD_PullupResistor pullup_resistor,
    SVC_DevGpioNotifyProxy proxy,
    SVC_DevNotifyClosure closure) ;
```

pin
The GPIO pin number of the pin to be configured as an input.

intr_config
The transitions of the input pin which are to generate an interrupt.

disarm_on_active
A boolean value indicating whether the pin interrupt is to be disarmed when it occurs.

pullup_resistor
A value for any pull up resistor to be associated with the pin pad. *N.B.* for pin #20, the resistor is actually a pull-down resistor. For all other pins, the pad resistor serves as a pull-up resistor.

proxy
A pointer to the background proxy function which is to be executed when the background notification associated with the pin interrupt is dispatched.

closure
A caller supplied data value which is passed along in the background notification.

The `dev_gpio_in_config` function configures the GPIO `pin` to be an input pin. The function returns zero on success and a negative error number otherwise.

```
<<dev realm gpio param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevGpioInConfigInput,
    GPIN_IntrConfig intr_config ;
    bool disarm_on_active ;
    PAD_PullupResistor pullup_resistor ;
    SVC_DevGpioNotifyProxy proxy ;
    SVC_DevNotifyClosure closure ;
) ;
```

```
<<dev svc gpio req: external definitions>>=
int
dev_gpio_in_config(
    SVC_DevInstance pin,
    GPIN_IntrConfig intr_config,
    bool disarm_on_active,
    PAD_PullupResistor pullup_resistor,
    SVC_DevGpioNotifyProxy proxy,
    SVC_DevNotifyClosure closure)
{
    SVC_DevGpioInConfigInput input = {
        .block_size = sizeof(SVC_DevGpioInConfigInput),
        .intr_config = intr_config,
        .disarm_on_active = disarm_on_active,
        .pullup_resistor = pullup_resistor,
        .proxy = proxy,
        .closure = closure,
    } ;
```

```
    SVC_DevRequest gpio_req = dev_req_encode(DEV_GPIO_CLASS, gpio_in_config, pin) ;
    return dev_realm_svc_call(gpio_req, &input, NULL, NULL) ;
}
```

```
<<dev realm gpio proxy: static data definitions>>=
static struct {
    bool disarm_on_active ;
    SVC_DevGpioNotifyProxy proxy ;
    SVC_DevNotifyClosure closure ;
} gpio_notifications[GPIO_PIN_INSTANCES] ;
```

```
<<dev realm gpio proxy: static function definitions>>=
static int
dev_realm_gpio_in_config(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevGpioInConfigInput gpio_input ;
    int status = copy_in_svc_param(input, sizeof(gpio_input), &gpio_input) ;
    rtcheck_zero_return(status, status) ;

    SVC_DevInstance pin = dev_req_extract_instance(req) ;
    rtcheck_max_return(pin, GPIO_PIN_INSTANCES, -ERR_INVALID_PARAM) ;
    if ((gpio_input.intr_config == gpin_intr_none && gpio_input.proxy != NULL) ||
            gpio_input.proxy == NULL) {
        return -ERR_INVALID_PARAM ;
    }

    status = gpio_pin_alloc(pin) ;
    rtcheck_zero_return(status, status) ;
    gpio_pin_intr_disable(pin) ;

    unsigned pad_offset ;
    uint32_t volatile *const pad_cfg_reg = gpio_pad_reg(pin, &pad_offset) ;
    uint32_t pad_cfg = *pad_cfg_reg ;

    unsigned cfg_offset ;
    uint32_t volatile *const cfg_reg = gpio_cfg_reg(pin, &cfg_offset) ;
    uint32_t cfg = *cfg_reg ;

    pad_cfg = btwd_bit_set(pad_cfg, pad_offset + PADREG_INPEN_OFFSET) ;
    pad_cfg = btwd_bits_insert(pad_cfg, PADREG_FNCSEL_GPIO,
            PADREG_FNCSEL_WIDTH, pad_offset + PADREG_FNCSEL_OFFSET) ;

    status = gpio_config_pullup_resistor(pin, gpio_input.pullup_resistor,
            pad_cfg, pad_offset, &pad_cfg) ;
    rtcheck_zero_return(status, status) ;

    cfg = btwd_bits_insert(cfg, 0, CFG_OUTCFG_WIDTH,
            cfg_offset + CFG_OUTCFG_OFFSET) ;
    switch (gpio_input.intr_config) {
    case gpin_intr_none:
        cfg = btwd_bit_clear(cfg, cfg_offset + CFG_INTD_OFFSET) ;
        cfg = btwd_bit_set(cfg, cfg_offset + CFG_INCFG_OFFSET) ;
        break ;
```

```
    case gpin_intr_both:
        cfg = btwd_bit_set(cfg, cfg_offset + CFG_INTD_OFFSET) ;
        cfg = btwd_bit_set(cfg, cfg_offset + CFG_INCFG_OFFSET) ;
        break ;

    case gpin_intr_low_high:
        cfg = btwd_bit_clear(cfg, cfg_offset + CFG_INTD_OFFSET) ;
        cfg = btwd_bit_clear(cfg, cfg_offset + CFG_INCFG_OFFSET) ;
        break ;

    case gpin_intr_high_low:
        cfg = btwd_bit_set(cfg, cfg_offset + CFG_INTD_OFFSET) ;
        cfg = btwd_bit_clear(cfg, cfg_offset + CFG_INCFG_OFFSET) ;
        break ;
    }

    GPIO->PADKEY = GPIO_PADKEY_PADKEY_Key ;
    *pad_cfg_reg = pad_cfg ;
    *cfg_reg = cfg ;
    GPIO->PADKEY = 0 ;

    if (gpio_input.intr_config != gpin_intr_none) {
        gpio_notifications[pin].disarm_on_active = gpio_input.disarm_on_active ;
        gpio_notifications[pin].proxy = gpio_input.proxy ;
        gpio_notifications[pin].closure = gpio_input.closure ;
        gpio_pin_intr_enable(pin) ;
    }

    return 0 ;
}
```

Disable a GPIO Pin

```
<<dev svc gpio req: external declarations>>=
extern int
dev_gpio_disable(
    SVC_DevInstance pin) ;
```

pin
The GPIO pin number of the pin to be disabled.

The `dev_gpio_disable` function disables `pin` and frees `pin` to be reused. Callers should disable any pins for which they have no further use to minimize power consumption.

```
<<dev svc gpio req: external definitions>>=
int
dev_gpio_disable(
    SVC_DevInstance pin)
{
    SVC_DevRequest gpio_req = dev_req_encode(DEV_GPIO_CLASS, gpio_disable, pin) ;
    return dev_realm_svc_call(gpio_req, NULL, NULL, NULL) ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static int
dev_realm_gpio_disable(
```

```
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(input == NULL) ; (void)input ;
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance pin = dev_req_extract_instance(req) ;
    rtcheck_max_return(pin, GPIO_PIN_INSTANCES, -ERR_INVALID_PARAM) ;

    gpio_pin_intr_disable(pin) ;
    gpio_notifications[pin].proxy = NULL ;
    gpio_pin_disable(pin) ;
    gpio_pin_free(pin) ;

    return 0 ;
}
```

Write an Output Pin

Since each pin can be represented by a single bit, rather than having a separate request for each output operation, this design uses a single function with a parameter which states the operation to perform.

```
<<dev realm gpio param: data type declarations>>=
typedef enum {
    gpio_pin_set_op,
    gpio_pin_clear_op,
    gpio_pin_toggle_op,
    gpio_tristate_set_op,
    gpio_tristate_clear_op,
    gpio_tristate_toggle_op,

    gpio_pin_op_COUNT               // last
} GPIO_PinOp ;
```

The set, clear, and toggle operations are conventional and control the output seen at the pin. For those output pins which have been configured as tri-state pins, the ability to control the tri-state setting is also given set, clear, and toggle operations on the tri-state setting.

```
<<dev svc gpio req: external declarations>>=
extern int
dev_gpio_write(
    SVC_DevInstance pin,
    GPIO_PinOp operation) ;
```

pin
: The GPIO pin number of the pin to be written.

operation
: The operation to perform on `pin`.

The `dev_gpio_write` function performs the operation indicated by, `operation`, on the pin given by, `pin`. The function returns zero on success and a negative error number otherwise.

```
<<dev realm gpio param: data type declarations>>=
```

```
DECLARE_REQUEST_PARAM(SVC_DevGpioWriteInput,
    GPIO_PinOp operation ;
) ;
```

```
<<dev svc gpio req: external definitions>>=
int
dev_gpio_write(
    SVC_DevInstance pin,
    GPIO_PinOp operation)
{
    SVC_DevGpioWriteInput input = {
        .block_size = sizeof(SVC_DevGpioWriteInput),
        .operation = operation,
    } ;

    SVC_DevRequest gpio_req = dev_req_encode(DEV_GPIO_CLASS, gpio_write, pin) ;
    return dev_realm_svc_call(gpio_req, &input, NULL, NULL) ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static int
dev_realm_gpio_write(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance pin = dev_req_extract_instance(req) ;
    rtcheck_max_return(pin, GPIO_PIN_INSTANCES, -ERR_INVALID_PARAM) ;

    SVC_DevGpioWriteInput gpio_input ;
    int status = copy_in_svc_param(input, sizeof(gpio_input), &gpio_input) ;
    rtcheck_zero_return(status, status) ;
    rtcheck_max_return(gpio_input.operation, gpio_pin_op_COUNT, -ERR_INVALID_PARAM) ;

    static void (*const pin_ops[gpio_pin_op_COUNT])(unsigned) = {       // ❶
        [gpio_pin_set_op] = gpio_pin_set,
        [gpio_pin_clear_op] = gpio_pin_clear,
        [gpio_pin_toggle_op] = gpio_pin_toggle,
        [gpio_tristate_set_op] = gpio_tristate_set,
        [gpio_tristate_clear_op] = gpio_tristate_clear,
        [gpio_tristate_toggle_op] = gpio_tristate_toggle,
    } ;

    assert(pin_ops[gpio_input.operation] != NULL) ;
    pin_ops[gpio_input.operation](pin) ;

    return 0 ;
}
```

❶ One of those awkward declarations in “C”. `pin_ops` is a `static` array of constant pointers to functions which take an `unsigned` argument and return no value. All of that for a simple jump table to the basic pin operations.

Read an Input Pin

```
<<dev svc gpio req: external declarations>>=
extern int
dev_gpio_read(
    SVC_DevInstance pin,
    unsigned *const input) ;
```

pin
The GPIO pin number of the pin to be read.

input
A pointer to an unsigned variable where the value of the pin is placed.

The `dev_gpio_read` function reads the current input value from `pin` and returns it by reference using the `input` pointer. The function returns zero on success and a negative error number otherwise. If an error occurs, `input` is not accessed.

```
<<dev realm gpio param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevGpioReadOutput,
    unsigned input ;
) ;
```

```
<<dev svc gpio req: external definitions>>=
int
dev_gpio_read(
    SVC_DevInstance pin,
    unsigned *const input)
{
    rtcheck_not_NULL_return(input, -ERR_INVALID_PARAM) ;

    SVC_DevGpioReadOutput output = {
        .block_size = sizeof(SVC_DevGpioReadOutput),
        .input = 0,
    } ;

    SVC_DevRequest gpio_req = dev_req_encode(DEV_GPIO_CLASS, gpio_read, pin) ;
    int status = dev_realm_svc_call(gpio_req, NULL, &output, NULL) ;

    if (status == 0) {
        *input = output.input ;
    }

    return status ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static int
dev_realm_gpio_read(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(input == NULL) ; (void)input ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance pin = dev_req_extract_instance(req) ;
    rtcheck_max_return(pin, GPIO_PIN_INSTANCES, -ERR_INVALID_PARAM) ;

    SVC_DevGpioReadOutput gpio_output = {
        .block_size = sizeof(SVC_DevGpioReadOutput),
```

```
        .input = gpio_pin_read(pin),
    } ;

    return copy_out_svc_result(output, sizeof(gpio_output), &gpio_output) ;
}
```

Arm a GPIO Pin

Part of the configuration for an input pin is whether it is to be disarmed when it is triggered. This request re-arms the pin when the application has decided it is ready to receive another input from the pin.

```
<<dev svc gpio req: external declarations>>=
extern int
dev_gpio_arm(
    SVC_DevInstance pin) ;
```

pin
The GPIO pin number of the pin to be armed.

The `dev_gpio_arm` function readies the input pin given by, `pin`, to notify changes in state. The function returns zero on success and a negative error number otherwise.

```
<<dev svc gpio req: external definitions>>=
int
dev_gpio_arm(
    SVC_DevInstance pin)
{
    SVC_DevRequest gpio_req = dev_req_encode(DEV_GPIO_CLASS, gpio_arm, pin) ;
    return dev_realm_svc_call(gpio_req, NULL, NULL, NULL) ;
}
```

```
<<dev realm gpio proxy: static function definitions>>=
static int
dev_realm_gpio_arm(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(input == NULL) ; (void)input ;
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance pin = dev_req_extract_instance(req) ;
    rtcheck_max_return(pin, GPIO_PIN_INSTANCES, -ERR_INVALID_PARAM) ;
    rtcheck_not_NULL_return(gpio_notifications[pin].proxy, -ERR_OPERATION_FAILED) ;

    gpio_pin_intr_enable(pin) ;

    return 0 ;
}
```

Dispatching GPIO Requests

Following our established pattern, a dispatch function for GPIO requests is constructed.

```
<<dev realm gpio proxy: external declarations>>=
extern int
dev_realm_gpio(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<dev realm gpio proxy: external function definitions>>=
int
dev_realm_gpio(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    int status = dev_req_validate(req, gpio_operation_count, GPIO_PIN_INSTANCES) ;
    rtcheck_zero_return(status, status) ;

    static SVC_DevRequestProxy const gpio_proxies[gpio_operation_count] = {
        [gpio_out_config] = dev_realm_gpio_out_config,
        [gpio_in_config] = dev_realm_gpio_in_config,
        [gpio_disable] = dev_realm_gpio_disable,
        [gpio_write] = dev_realm_gpio_write,
        [gpio_read] = dev_realm_gpio_read,
        [gpio_arm] = dev_realm_gpio_arm,
    } ;

    SVC_DevOperation gpio_operation = dev_req_extract_operation(req) ;
    return gpio_proxies[gpio_operation](req, input, output, error) ;
}
```

```
<<dev realm param: constants>>=
#define DEV_GPIO_CLASS        3
```

```
<<svc entry: device request classes>>=
[DEV_GPIO_CLASS] = dev_realm_gpio,
```

GPIO IRQ Handler

The IRQ handler for the GPIO pins has to find which pins are currently interrupting. At any given time, that is most probably only 1 pin. The interrupt information about the pins is in separate register arrays. With 50 pins and 32 bit registers, it takes two array elements to hold all the information. The layout of the registers is in the form of a structure, composed into an array of two. No matter, two nested loops are used. The first loop iterates over the controls which are grouped by 32 pins. So, there is at most two times through the outer loop. The inner loop handles at most 32 pins, matching encoding of pins in a 32-bit register.

```
<<dev realm gpio proxy: external function definitions>>=
void
GPIO_IRQHandler(void)
{
    GPIO_INTCTRL_Type volatile *intctrl = GPIO->INTCTRL ;
    GPIO_INTCTRL_Type volatile *const end = GPIO->INTCTRL + COUNTOF(GPIO->INTCTRL) ;

    unsigned pin_offset = 0 ;                                    // ❶
    for ( ; intctrl < end ; intctrl++) {
        uint32_t int_pending = intctrl->INTSTAT & intctrl->INTEN ;
        while (int_pending != 0) {
```

```
            uint8_t lead_bit = 31U - __CLZ(int_pending) ;           // ❷
            uint32_t mask = btwd_bit_mask(lead_bit) ;

            <<gpio irq handler: service one pin>>

            int_pending &= ~mask ;
        }

        pin_offset += 32 ;
    }
}
```

❶ The first 32 pins are handled by the first iteration and the remaining 18 pins are accounted for in the second iteration of the loop.

❷ Counting the leading zeros in the status register can be used to determine the pin number causing an interrupt. This saves examining the status register bit by bit in a loop, only to find that most of the pins are not interrupting at this moment. *N.B.* this services the pins in descending order of pin number. Servicing in ascending pin order can be achieved by reversing the bits in the `int_pending` variable using the `__RBIT()` CMSIS function.

Servicing a single pin clears the interrupt and then looks up the background proxy function to which the notification is sent.

```
<<gpio irq handler: service one pin>>=
intctrl->INTCLR = mask ;
unsigned pin = pin_offset + lead_bit ;

if (gpio_notifications[pin].disarm_on_active) {                 // ❶
    gpio_pin_intr_disable(pin) ;
}
SVC_DevGpioNotifyProxy proxy = gpio_notifications[pin].proxy ;
if (proxy != NULL) {
    <<gpio irq hander: send background notification>>
} else {
    gpio_pin_intr_disable(pin) ;
}
```

❶ This is where disarming happens for pins so configured.

Having found a proxy function, sending the notification follows the usual steps. Notice in this case that the notification is built into the allocated space directly before it is "pushed" to the background notification queue. This works best when the potential to build several notifications exists.

```
<<gpio irq hander: send background notification>>=
SVC_DevGpioNotification *notification =
        probe_bg_queue(sizeof(SVC_DevGpioNotification)) ;
if (notification == NULL) {
    panic("failed to probe GPIO background notification: pin = %u", pin) ;
}

notification->block_size = sizeof(SVC_DevGpioNotification) ;
notification->notify_proxy = (SVC_DevNotifyProxy)proxy ;
notification->notify_closure = gpio_notifications[pin].closure ;
notification->device_class = DEV_GPIO_CLASS,
notification->device_instance = pin ;
notification->pin_value = gpio_pin_read(pin) ;

bool pushed = push_bg_queue() ;
if (!pushed) {
    panic("failed to push GPIO background notification: pin = %u", pin) ;
}
```

Code Layout

GPIO service requests

```
<<dev_svc_gpio_req.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Function prototypes for GPIO service requests.
 *--
 */
#ifndef DEV_SVC_GPIO_REQ_H_
#define DEV_SVC_GPIO_REQ_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "dev_svc_req.h"
#include "dev_realm_gpio_param.h"
/*
 * Data Type Declarations
 */
<<dev svc gpio req: data type declarations>>
/*
 * External Declarations
 */
<<dev svc gpio req: external declarations>>

#endif /* DEV_SVC_GPIO_REQ_H_ */
```

GPIO Device Service Requests

```
<<dev_svc_gpio_req.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Device Realm GPIO Request Implementation
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "useful.h"
#include "rtcheck.h"
```

```
#include "svc_req_errors.h"
#include "svc_req.h"
#include "dev_svc_req.h"
#include "dev_svc_gpio_req.h"
#include "dev_realm_gpio_param.h"
/*
 * External Functions
 */
<<dev svc gpio req: external definitions>>
```

Device Realm GPIO Parameters

```
<<dev_realm_gpio_param.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Parameter interface data structures for GPIO device requests.
 *--
 */
#ifndef DEV_REALM_GPIO_PARAM_H_
#define DEV_REALM_GPIO_PARAM_H_
/*
 * Include Files
 */
#include "dev_realm_param.h"
#include "apollo3.h"
/*
 * Constants
 */
<<dev realm gpio param: constants>>
/*
 * Static Inline Functions
 */
<<dev realm gpio param: static inline functions>>
/*
 * Data Type Declarations
 */
<<dev realm gpio param: data type declarations>>

#endif /* DEV_REALM_GPIO_PARAM_H_ */
```

Device Realm GPIO Proxy Header

```
<<dev_realm_gpio_proxy.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Interfaces for device realm gpio proxy functions.
```

```
 *--
 */
#ifndef DEV_REALM_GPIO_PROXY_H_
#define DEV_REALM_GPIO_PROXY_H_

/*
 * Include files
 */
#include "dev_realm_param.h"
/*
 * External Declarations
 */
<<dev realm gpio proxy: external declarations>>

#endif /* DEV_REALM_GPIO_PROXY_H_ */
```

```
<<svc handler: device include files>>=
#include "dev_realm_gpio_proxy.h"
```

Device Realm GPIO Proxy Code Layout

```
<<dev_realm_gpio_proxy.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for device realm GPIO proxy functions.
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "svc_req_errors.h"
#include "svc_proxy.h"
#include "dev_realm_proxy.h"
#include "dev_realm_gpio_param.h"
#include "dev_realm_gpio_proxy.h"
#include "bg_req_queue.h"
#include "panic.h"
#include "bit_twiddle.h"
#include "sys_twiddle.h"
#include "useful.h"
#include "rtcheck.h"
<<dev realm gpio proxy: include files>>
/*
 * Constants
 */
<<dev realm gpio proxy: constants>>
/*
 * Data Type Declarations
 */
<<dev realm gpio proxy: data type declarations>>
/*
 * Forward References
```

```
 */
<<dev realm gpio proxy: forward references>>
/*
 * Static Data Definitions
 */
<<dev realm gpio proxy: static data definitions>>
/*
 * Static Inline Function Definitions
 */
<<dev realm gpio proxy: static inline functions>>
/*
 * Static Function Definitions
 */
<<dev realm gpio proxy: static function definitions>>
/*
 * External Function Definitions
 */
<<dev realm gpio proxy: external function definitions>>
```

Examples

In this chapter, four examples of using GPIO pins are shown.

Blinky

In the microcontroller world, a program to blink an LED carries a similar status as the *hello, world* program in the conventional timeshare world. The LED is controlled by an output pin. As shown previously, the built-in LED is attached to GPIO pin number 19.

```
<<blinky: main>>=
#define LED_BUILTIN     19U
```

The usual blinky program design uses blocking timing to alternate turning on and then off the LED. This implementation is quite different. In this design, there is no blocking timing mechanism. The timer queue requests developed previously are used.

The `main` program first configures the output pin for the LED and sets up a periodic timer for 500 ms. Upon each timer expiration, the value of the output pin is toggled. This effectively generates a 1 second square wave with a 50% duty cycle. What you see is the LED alternately turn of for 500 ms and then turn off for 500 ms.

```
<<blinky: main>>=
void
main(void)
{
    int status ;

    status = dev_gpio_out_config(LED_BUILTIN, gpout_pushpull,
            pad_drive_12mA, pad_pullup_none) ;
    assert(status == 0) ; (void)status ;

    status = dev_timq_open(0, (SVC_DevTimqNotifyProxy)svc_proxy_var_sync) ;
    assert(status == 0) ;

    bool volatile expiration_done ;
    TIMQ_TimeTicks const ticks_500ms = dev_util_timq_ms_to_ticks(500) ;
    TIMQ_ElementID elemID = dev_timq_insert(0, ticks_500ms, ticks_500ms,
            (SVC_DevNotifyClosure)&expiration_done) ;
    assert(elemID >= 0) ; (void)elemID ;
```

```
    puts("blinking start") ;

    for (int i = 0 ; i < 30 ; i++) {                // ❶
        status = dev_gpio_write(LED_BUILTIN, gpio_pin_toggle_op) ;
        assert(status == 0) ;
        puts("toggle") ;

        sys_ctrl_busy_wait(&expiration_done) ;
    }

    status = dev_timq_close(0) ;
    assert(status == 0) ;
    sys_ctrl_busy_wait(&expiration_done) ;                    // ❷

    status = dev_gpio_disable(LED_BUILTIN) ;
    assert(status == 0) ;

    puts("blinking end") ;
}
```

❶ An arbitrary number so the example terminates at some point.

❷ Since the timer was configured as periodic, a notification is sent when the timer queue is closed. Although not strictly necessary for this case, the program waits for the notification to demonstrate how to shut down the timer queue gracefully.

Code Layout

```
<<tyranny-of-the-pins-blinky-test.c>>=
<<edit warning>>
<<copyright info>>

#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include "sys_svc_req.h"
#include "dev_svc_req.h"
#include "dev_svc_timq_req.h"
#include "dev_svc_gpio_req.h"

<<test util: forward references>>
<<blinky: main>>
<<test util: static functions>>
```

Button

Momentary contact switches are a common input device for microcontroller systems. This example demonstrates using a GPIO pin as an input to detect button pressing. The following diagram shows an external circuit used to interface the button to a GPIO pin. The particular pin chosen is for convenience in hooking up the circuit to the processor pin.

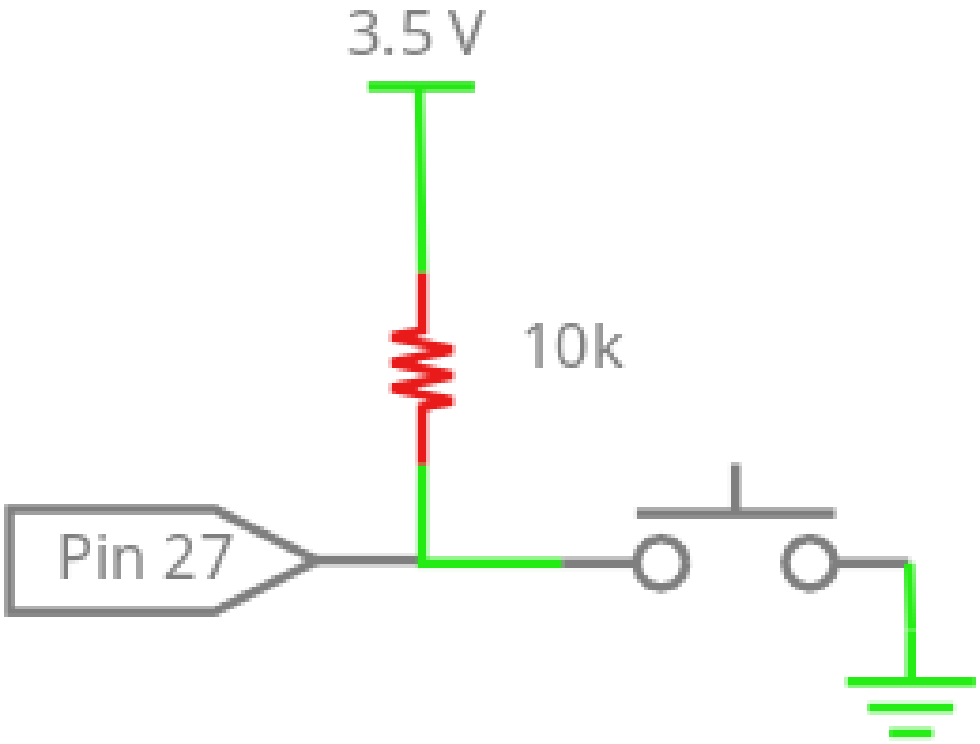

Figure 8.4: Push Button Circuit

This is a simple electrical circuit. Power is connected to a switch via a 10 kOhm resistor. The resistor insures that power is not shorted to ground when the button is pressed.

```
<<button: main>>=
#define BUTTON_PIN     27U

void
main(void)
{
    bool volatile button_pressed ;
    unsigned pressed_value ;

    int status = dev_gpio_in_config(BUTTON_PIN, gpin_intr_both,
            false, pad_pullup_none, (SVC_DevGpioNotifyProxy)svc_proxy_var_sync,
            (SVC_DevNotifyClosure)&button_pressed) ;
    assert(status == 0) ; (void)status ;

    dev_gpio_read(BUTTON_PIN, &pressed_value) ;
    printf("button value = %d\n", pressed_value) ;
    printf("press button!\n") ;

    for (int i = 0 ; i < 30 ; i++) {
        sys_ctrl_busy_wait(&button_pressed) ;
        printf("got button: %u\n", pressed_value) ;
    }

    status = dev_gpio_disable(BUTTON_PIN) ;
    assert(status == 0) ;

    printf("button press completed\n") ;
}
```

Code Layout

```
<<tyranny-of-the-pins-button-test.c>>=
<<edit warning>>
<<copyright info>>

#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include "sys_svc_req.h"
#include "dev_svc_req.h"
#include "dev_svc_gpio_req.h"

<<test util: forward references>>
<<button: main>>
<<test util: static functions>>
```

In and Out

GPIO pins are an extraordinarily useful means to provide insight into a running program with minimal effect on the program's execution. The usual strategy is to change the state of an I/O pin and measure the outcome using an external device such as a logic analyzer or oscilloscope.

In this example, the cycle time is measured from taking an interrupt on an input pin through the background notification and then back to the foreground to toggle an output pin. A waveform generator is attached to an input pin and the generator is configured to create a square wave. By measuring the waveform created on the output pin, a *phase* shift is observed between the rising edge of the input and the rising edge of the output. The following diagram shows a simplified timing diagram of what the example does.

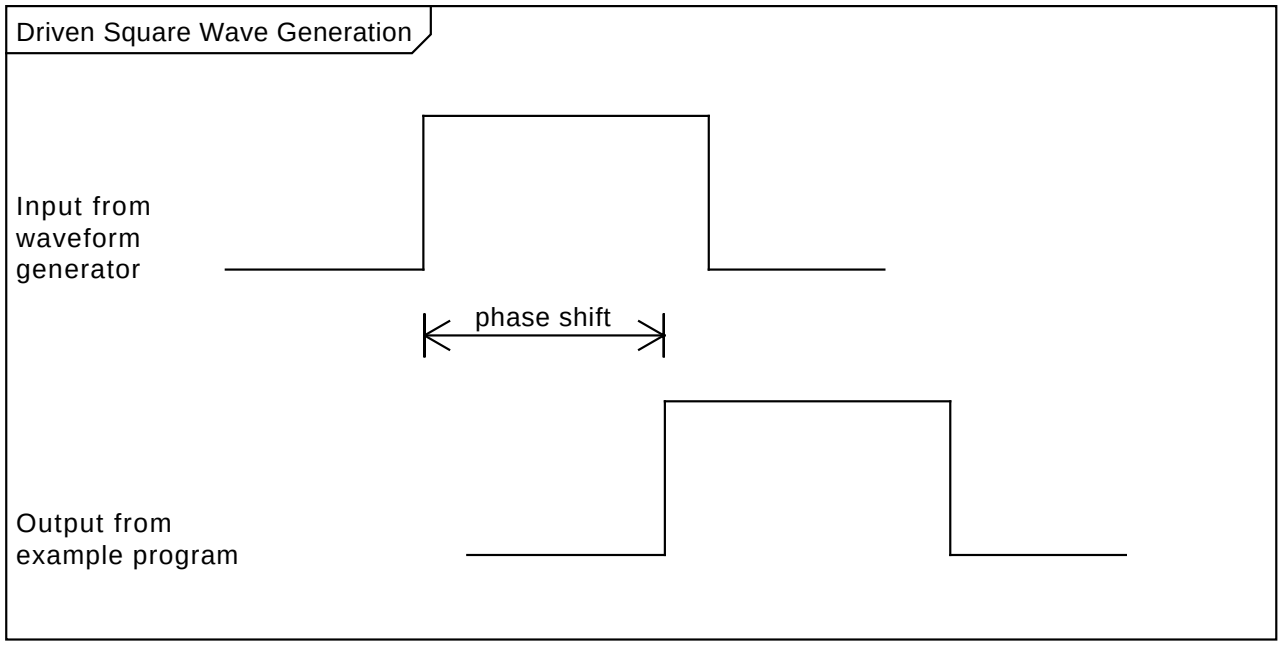

Figure 8.5: GPIO Example Timing Diagram

The amount of time between corresponding rising edges gives us a measure of the time required to process:

1. the input pin IRQ
2. queue a background notification
3. retrieve and dispatch the notification
4. process a background request to change the state of the output pin.

The choice of GPIO pins to use in the example is for convenience. In this case, pins 14 and 15 are used.

```
<<in and out: main>>=
#define WF_IN_PIN       14U     /* micro mod GPIO 6 */
#define GEN_OUT_PIN     15U     /* micro mod GPIO 7 */
```

Pin 14 is configured as an input which is triggered on both edges. Pin 15 is configured as an output. The background notification function for pin 14 simply requests immediately that pin 15 be toggled.

```
<<in and out: main>>=
static void
in_triggered(
    SVC_DevGpioNotification const *const notify)
{
    dev_gpio_write(GEN_OUT_PIN, gpio_pin_toggle_op) ;
}
```

The `main` program configures the GPIO pins and runs an infinite event loop.

```
<<in and out: main>>=
void
main(void)
{
    int status = dev_gpio_in_config(WF_IN_PIN, gpin_intr_both,
            false, pad_pullup_none, in_triggered, 0) ;
    assert(status == 0) ; (void)status ;

    status = dev_gpio_out_config(GEN_OUT_PIN, gpout_pushpull,
            pad_drive_2mA, pad_pullup_none) ;
    assert(status == 0) ;

    bool volatile forever ;
    sys_ctrl_busy_wait(&forever) ;
}
```

The following figure shows the results of the waveform generated on the output pin as the example program attempts to mimic the changes in the input pin state. The code was built in release mode to achieve best performance. The waveform generator driving the input is running at 5 kHz. The phase shift between the input waveform and the output is approximately 143 μs

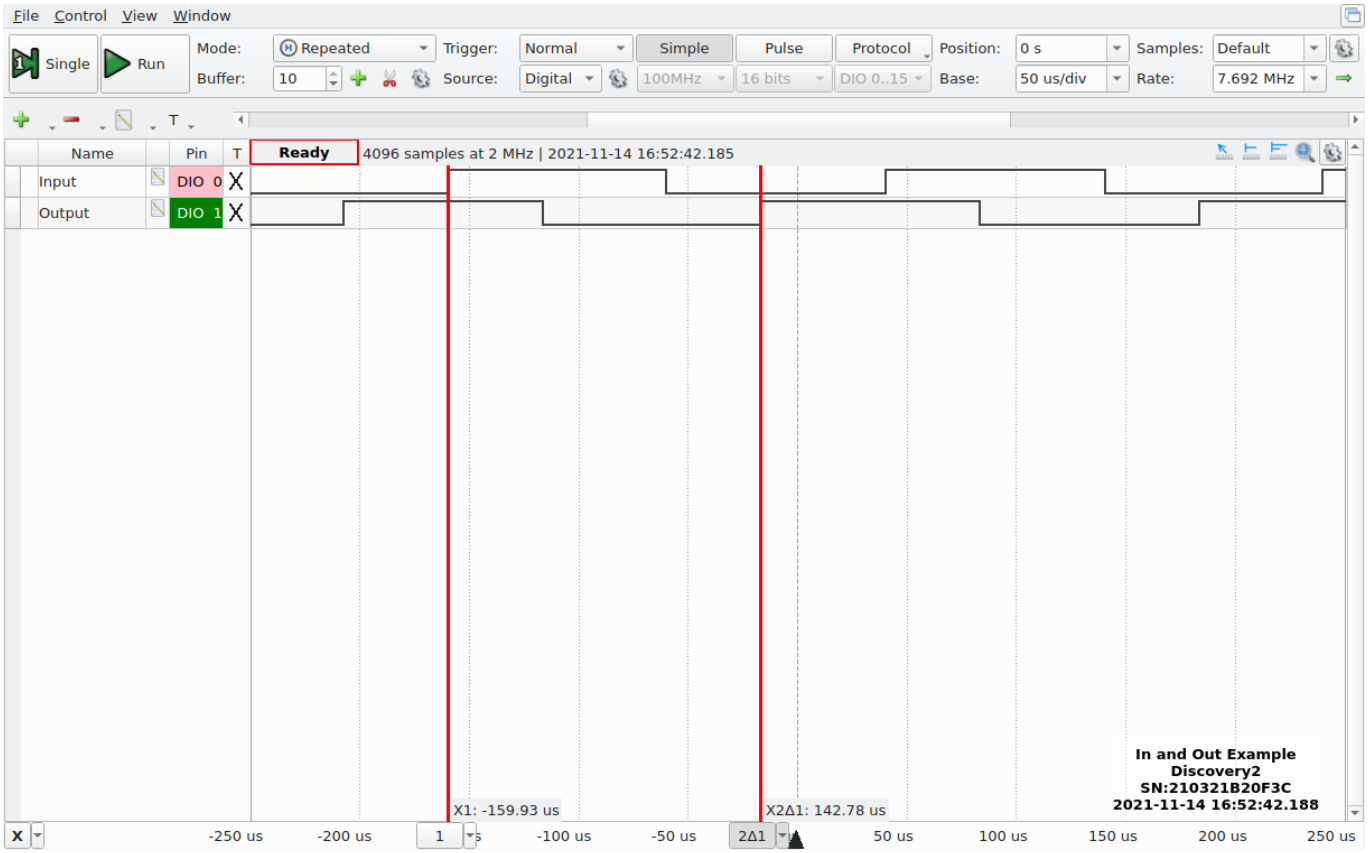

Figure 8.6: Waveform Generation Results

Taking a conservative value of 150 μs as the turn around time, then the maximum frequency input waveform is approximately 6.7 kHz. This sets the lower time bound that the system can execute a response in background processing to a change in the system environment.

It is interesting to consider what happens to this program if the input generated waveform is greater than 6.7 kHz, say 10 kHz. In that case, background notifications cannot be removed and processed from the background notification queue quickly enough and, in a short time, the background notification queue overflows. As the code is written, the GPIO IRQ handler detects the overflow and treats it as a **panic** condition.

Code Layout

```
<<tyranny-of-the-pins-in-out-test.c>>=
<<edit warning>>
<<copyright info>>

#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include "sys_svc_req.h"
#include "dev_svc_gpio_req.h"

<<in and out: main>>
```

Button Blinky

The last GPIO example combines blinking an LED under the control of a push button. The resulting program uses the push button to start and stop blinking of the LED. When the button is pressed, the LED is blinked at a 1 Hz rate. When pressed again, the LED blinking is stopped. Each press of the button, toggles the 1 Hz blinking on and off.

The implementation of this example is expanded considerably. The approach presented is similar to that which might be used for larger scale background execution. As mentioned previously, up to now how background execution operates has not been described. As a preview, the background execution model uses finite state machines as the primary mechanism to sequence execution. This example also uses state machines, but with considerably simplified techniques. The example serves as an introduction to the background execution model used when building larger applications. The goal here is to establish the basic rules for the type of state machine execution that are used later.

A simplified state model execution scheme is used that is synchronously executed by the program. There is no event loop and once an event is signaled from outside of the machine, the resulting transitions and actions run to completion. The mechanism is self-contained and also suitable to be used from an IRQ handler in the sense that there are no blocking operations. The state models are Moore type machines. The mechanism extends the formal Moore machine definition by executing an action, in this case of a body of "C" code, when the state is entered[2]. This implies that computation is directly associated with the transition into a state. All actions execute to completion before the consequence of any events received or sent are considered. Thus, if a state action signals an event to another state machine as part of a state action, execution continues and the dispatch of the signaled event occurs after the state action completes. Events do *not* carry any parameters. Events directly signaled by a state machine to itself are handled before any events signaled from a different state machine. There are no *guard* expressions, hierarchy or other more complex formulations of state model behavior. One state machine may directly signal an event another machine and a state machine may signal itself an event to implement transitory states. The set of state machine rules given here is adequate for simpler situations. Later, these rules are expanded when the background execution model is described. For now, the simpler rules allow for easier reasoning about what the state behavior is.

Blinky State Model

The diagram below is a state model diagram for blinking the LED. The graphical notation is a restricted subset of UML state diagram notation. Each yellow box represents a state. Arrows represent events and are labeled by an event name. An arrow connecting two boxes implies a state transition when event is received. The arrow pointing from the black circle to a yellow box indicates the default initial state of the model. When a state machine is created, it is placed in the initial state *without* executing the associated action. Only when the machine transitions, is the action contained in the state box executed. The actions are described by *pseudo-code* statements which express the intent of the action's computation.

[2] UML nomenclature for this is an *entry action*. Only entry actions are used here and no further classification of actions is considered

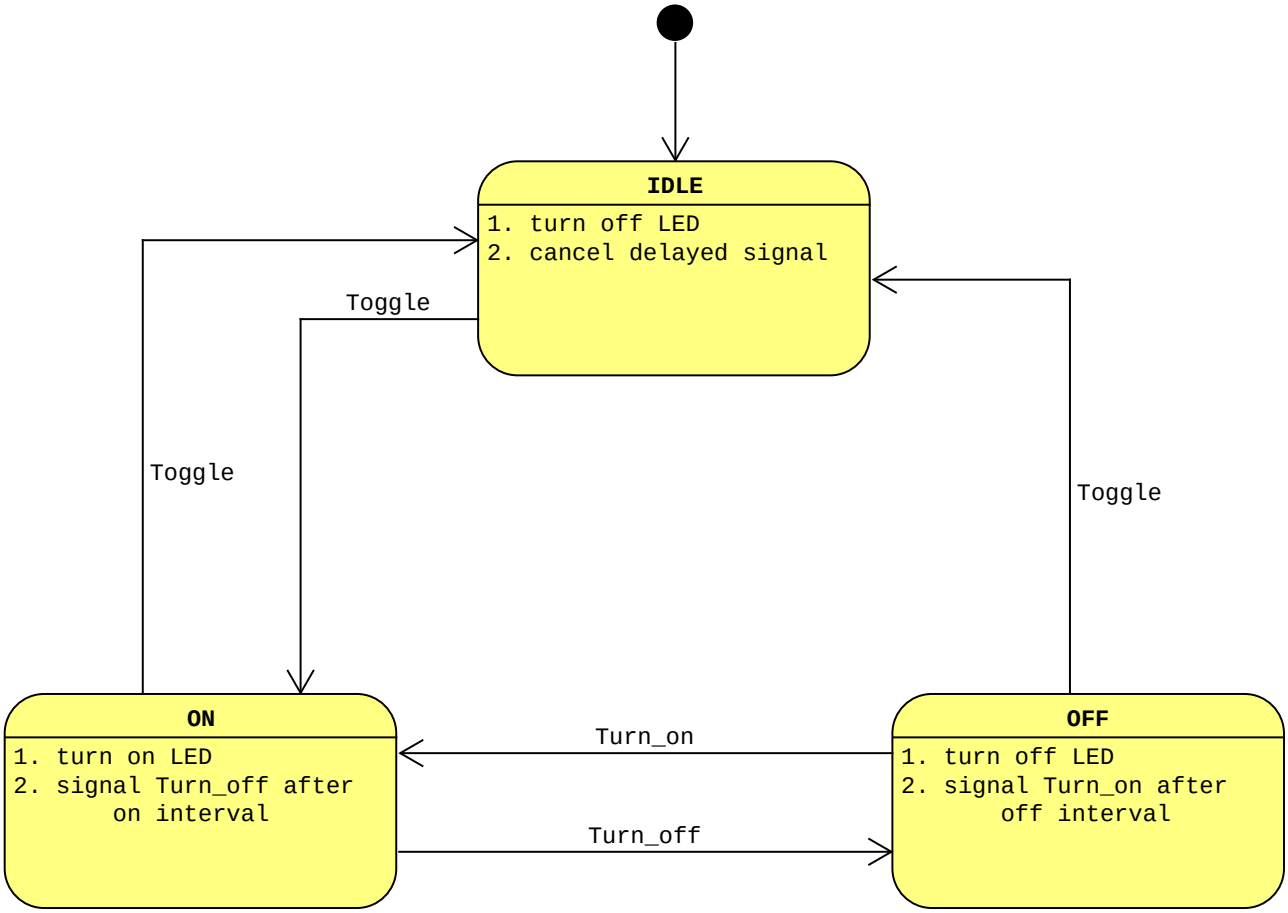

Figure 8.7: Blinky State Model

For an LED, the state model starts in the **IDLE** state. Since no action is executed when the machine is created, it is assumed that initialization code has turned off the LED. In this case, the output pin is zero after reset. When the **Toggle** event is received, the transition to the **ON** state is made. The action of the **ON** state turns on the LED and arranges for the **Turn_off** event to be signaled at some time in the future. When the **Turn_off** event arrives the transition to the **OFF** state happens. The **OFF** state action turns off the LED and arranges for the **Turn_on** event to be signaled at some future time. The sum of the on and off intervals is contrived to be one second yielding a 1 Hz blink rate. This behavior continues repeating on/off, on/off, etc. as the state transitions continue to be driven by time delayed events. At some time, when the **Toggle** event is received, the machine transitions to the **IDLE** state where the action turns off the LED and cancels any delayed signal created by either the **ON** or **OFF** states.

Note, this is *not* the only way this behavior could be captured in a state model. In general, a given problem may have multiple state model solutions in much the same manner that it would have multiple directly programmed solutions. This model emphasizes the correspondence of the illumination state of the physical LED with the conceptual state of the model both of which are separate from the behavior to initiate and terminate a continuing blinking sequence. An alternative state model might have only two states and use the ability to toggle a GPIO pin value. That effectively holds part of the state of the LED pin value in the hardware.

To access the functions in our synchronous state machine module, the header file with the interface declarations must be included.

```
<<button blinky: include files>>=
#include "sync_sm.h"
```

In our formulation of finite state machines, a careful distinction is made between a *state model* and a *state machine*. A state model is used to describe the behavior shown in the previous diagram. A state machine is an instance of a state model associated

with some parametric data. All state machine instances of a given state model behave the same. However, each state machine instance has its own notion of its current state and access to any parametric data associated with the particular state machine. This implies there may be multiple state machines based on the same state model, all executing independently and generally in different states, yet the state machines go through the same transitions, exhibit the same behavior, and execute the same code.

Defining the Blinky State Model

The first step is to undertake encoding the behavior implied by the previous diagram. The synchronous state model module defines a number of pre-processor macro functions to save some typing and hide some of the details of the language mechanisms used to implement the model execution rules.

First, the states in the Blinky model are enumerated.

```
<<button blinky: state model>>=
typedef enum {
    leds_IDLE = 0,
    leds_ON,
    leds_OFF,

    leds_STATE_COUNT    // last
} Led_State ;
```

One important aspect of state machine execution is to obtain the chronological trace of event dispatch. To make that easier for humans to consume, define the names given to the states.

```
<<button blinky: state model>>=
static const
SSM_DEFINE_STATE_NAMES(blinky_model, leds_STATE_COUNT) = {
    SSM_NAME_STATE(leds_IDLE, IDLE),                        // ❶
    SSM_NAME_STATE(leds_ON, ON),
    SSM_NAME_STATE(leds_OFF, OFF),
} ;
```

❶ The second argument is the state name which is “stringified” by the macro.

Just like the states, define the events in the state model.

```
<<button blinky: state model>>=
typedef enum {
    lede_Toggle = 0,
    lede_Turn_on,
    lede_Turn_off,

    lede_EVENT_COUNT    // last
} Led_Event ;
```

```
<<button blinky: state model>>=
static const
SSM_DEFINE_EVENT_NAMES(blinky_model, lede_EVENT_COUNT) = {
    SSM_NAME_EVENT(lede_Toggle, Toggle),
    SSM_NAME_EVENT(lede_Turn_on, Turn_on),
    SSM_NAME_EVENT(lede_Turn_off, Turn_off),
} ;
```

The third major component of the state model is the activity table. The data in the activity table is used to invoke functions which contain the code associated with a state. The information required is a mapping from the enumerated value of a state to a pointer to a function which is the state activity.

```
<<button blinky: state model>>=
static const
SSM_DEFINE_ACTIVITY_TABLE(blinky_model, leds_STATE_COUNT) = {
    SSM_DEFINE_ACTIVITY(leds_IDLE, idle_activity),
    SSM_DEFINE_ACTIVITY(leds_ON, on_activity),
    SSM_DEFINE_ACTIVITY(leds_OFF, off_activity),
} ;
```

An activity function contains the code for the state and has an interface which returns nothing and accepts a single pointer parameter to the state machine for which the action is run. For the three states in the Blinky model, the action functions have the following prototype.

```
<<button blinky: forward references>>=
static void idle_activity(SSM_Machine *const self) ;
static void on_activity(SSM_Machine *const self) ;
static void off_activity(SSM_Machine *const self) ;
```

The last major component of a state model is the transition function. This function maps the current state and an event into a new target state. The new state may be any state defined by the model or one of two non-transitioning states. Here, two additional transition rules are introduced which are encoded as a non-transitioning target state.

- **IG** as a target state means that the event is ignored. Sometimes events simply arrive to late to be considered. For example, a state model describing a device which can be powered up and down does not care if a **Power down** event arrives when it is already powered off. The event simply arrived too late. Ignoring an event can be implemented by adding states to the model, but the need is so common that ignored states would significantly clutter the diagram.
- **CH** as a target state means that the event logically can't happen in the given state. If an event is received and the new target state is **CH**, no transition happens and it is considered an error condition. For example, in the Blinky state model, the **Turn_on** event cannot happen when a machine is in the **ON** state. This is because the event is only generated by the **OFF** state and it was the receipt of the **Turn_on** event while in the **OFF** state that causes the transition to the **ON** state. If the **Turn_on** event is received while in the **ON** state, something is terribly wrong. Can't happen transitions are usually the result of errors in problem analysis.

Note that the non-transitioning target states of **IG** and **CH** do *not* appear on the state model graphical diagram. Only the transitions to other states in the model show up in the diagram. This is because the non-transitioning states have little impact on the reasoning about the actions of the model and so adding them would contribute to diagram clutter. For these simple examples, the state model diagrams are equally simple. However, for state models larger to more than a handful of states, it is beneficial to reasoning about the behavior to keep the diagrams as simple as possible.

The transition function is usually encoded as a two-dimensional array of *states* number of rows and *events* number of columns. The following table show the transition matrix for the Blinky state model.

Table 8.1: Blinky Transition Matrix

	Toggle	**Turn_on**	**Turn_off**
IDLE	ON	IG ❶	IG ❶
ON	IDLE	CH	OFF
OFF	IDLE	ON	CH

❶ The **Turn_on** and **Turn_off** events are ignored in the **IDLE** state because there is a race condition for delivery of the delayed events. It is possible for the GPIO IRQ handler to queue a background notification and then for the Timer Queue IRQ handler to queue a time expiration before the pin notification can be dispatched and acted upon. Since the Timer Queue IRQ handler can preempt the state machine action execution, the background notification from the timer queue could already be in flight when the attempt to cancel the timer happens in the **IDLE** state. Again, this is just a case of the event arriving too late.

The definition of the transition function is a direct transliteration of the above transition matrix into pre-processor macro invocations.

```
<<button blinky: state model>>=
static const
SSM_DEFINE_TRANS_TABLE(blinky_model, leds_STATE_COUNT, lede_EVENT_COUNT) = {
    SSM_DEFINE_TRANSITION(leds_IDLE, lede_Toggle, leds_ON),
    SSM_DEFINE_TRANSITION(leds_IDLE, lede_Turn_on, SSM_State_IG),
    SSM_DEFINE_TRANSITION(leds_IDLE, lede_Turn_off, SSM_State_IG),

    SSM_DEFINE_TRANSITION(leds_ON, lede_Toggle, leds_IDLE),
    SSM_DEFINE_TRANSITION(leds_ON, lede_Turn_on, SSM_State_CH),
    SSM_DEFINE_TRANSITION(leds_ON, lede_Turn_off, leds_OFF),

    SSM_DEFINE_TRANSITION(leds_OFF, lede_Toggle, leds_IDLE),
    SSM_DEFINE_TRANSITION(leds_OFF, lede_Turn_on, leds_ON),
    SSM_DEFINE_TRANSITION(leds_OFF, lede_Turn_off, SSM_State_CH),
} ;
```

Blinky State Machine

Usually, the actions of a state model access other parametric data. This is accomplished by embedding the state machine in a `struct` which contains the required data. For the Blinky model, the state actions refer to an `on interval` and an `off interval`. These are time delays for when an event should be signaled. The timer queue requests are used to obtain the necessary delay times.

The following structure shows the layout of data needed by the state actions of the Blinky model.

```
<<button blinky: data type declarations>>=
typedef struct {
    SSM_Machine machine ;
    unsigned pin ;
    GPIO_PinOp pin_active_op ;
    GPIO_PinOp pin_inactive_op ;
    unsigned on_interval_ms ;
    unsigned off_interval_ms ;
    TIMQ_ElementID timer ;
} Blinky_Machine ;
```

machine
: The state machine that controls an individual LED. Note, the machine is *contained* within the parametric data structure and not access by reference. This member holds the current state of the machine and references to the state model which contains the transition function.

pin
: The GPIO pin number attached to the LED.

pin_active_op
: The operation to apply to `pin` to turn on the LED.

pin_inactive_op
: The operation to apply to `pin` to turn off the LED.

on_interval_ms
: The number of milliseconds the LED is to remain on before being switched off.

off_interval_ms
: The number of milliseconds the LED is to remain off before being switched on.

timer
: The ID of a timer queue element which is used to control the delay times for turning the LED on and off.

With the parametric data in hand, the code for the state activities can be constructed. Processing for the state is placed in a function whose prototype was shown earlier.

```
<<button blinky: state model>>=
static void
idle_activity(
    SSM_Machine *const self)
{
    Blinky_Machine *bm = CONTAINER_OF(self, Blinky_Machine, machine) ;  // ❶

    int status = dev_gpio_write(bm->pin, bm->pin_inactive_op) ;
    assert(status == 0) ; (void)status ;

    status = dev_timq_remove(0, bm->timer) ;
}
```

❶ The state machine dispatch code works with pointers to values of type, `SSM_Machine`. Here, the type of the enclosing structure is *recovered* with a pre-processor macro. Ultimately, the standard `offsetof` macro is used to perform some pointer arithmetic. This may seem dangerous since the compiler is no help in determining if the macro invocation is correct. Careful coding and context awareness are necessary.

```
<<button blinky: state model>>=
static void
on_activity(
    SSM_Machine *const self)
```

```
{
    Blinky_Machine *bm = CONTAINER_OF(self, Blinky_Machine, machine) ;

    int status = dev_gpio_write(bm->pin, bm->pin_active_op) ;
    assert(status == 0) ; (void)status ;

    bm->timer = dev_timq_insert(0,
            dev_util_timq_ms_to_ticks(bm->on_interval_ms), 0,
            (SVC_DevNotifyClosure)ds_blinky_Turn_off) ;          // ❶
    assert(bm->timer >= 0) ;
}
```

❶ An event is signaled after a delay by using the timer queue background notifications and passing a closure value to indicate the event to be signaled. This technique is discussed below.

```
<<button blinky: state model>>=
static void
off_activity(
    SSM_Machine *const self)
{
    Blinky_Machine *bm = CONTAINER_OF(self, Blinky_Machine, machine) ;

    int status = dev_gpio_write(bm->pin, bm->pin_inactive_op) ;
    assert(status == 0) ; (void)status ;
    bm->timer = dev_timq_insert(0,
            dev_util_timq_ms_to_ticks(bm->off_interval_ms), 0,
            (SVC_DevNotifyClosure)ds_blinky_Turn_on) ;
    assert(bm->timer >= 0) ;
}
```

```
<<button blinky: state model>>=
static
SSM_DEFINE_CONTEXT(button_blinky_context) ;
```

The last detail is to define a state machine execution *context*. In this example, one state machine signals another one. When multiple state machine interact, the event dispatch mechanisms require a context which is used to deliver events to the participating state machines.

```
<<button blinky: state model>>=
static const
SSM_DEFINE_STATE_MODEL(blinky_model, leds_IDLE,
        leds_STATE_COUNT, lede_EVENT_COUNT) ;
```

As before, the LED is connected to pin 19.

```
<<button blinky: constants>>=
#define LED_BUILTIN     19U
```

Finally, the state machine data structure that blinks the LED connected to pin 19 is defined. A variable named, `blinky`, is used to hold the required data for the machine.

```
<<button blinky: state model>>=
Blinky_Machine blinky = {
    SSM_MACHINE_INITIALIZER(blinky, machine, &blinky_model, leds_IDLE,
            &button_blinky_context),
    .pin = LED_BUILTIN,
    .pin_active_op = gpio_pin_set_op,
    .pin_inactive_op = gpio_pin_clear_op,
    .on_interval_ms = 200U,             // ❶
```

```
    .off_interval_ms = 800U,
    .timer = -1,
} ;
```

1. An LED surprisingly consumes quite a bit of current, as much as 15 - 20 mA. One way to minimize the power consumption is to modulate the on time to being short. You don't need a 50% duty cycle wave to see the LED come on.

Bouncy State Model

Momentary contact push buttons usually exhibit a phenomenon called, *bouncing*. Bouncing occurs when the metal plates of the switch come together and can cause the signal transition from inactive to active to have ripples in it. These ripples can be large enough for the input pin to detect them as multiple transitions. Combined with a spring that causes the button to release, the signal seen by the input pin is *not* the clean square wave on either its rising or falling edge. This certainly happens for our case as was demonstrated by the previous button example.

Noisy switch signals can be handled in a number of ways. Sometimes there are *glitch* filters provided on the I/O pin by the microcontroller. It is possible to use external filtering electronics to smooth the signal so that it appears like a square wave. However, frequently it falls on software to provide "debouncing" for push buttons.

There are many techniques to debounce a switch in software. Most techniques involve the idea of waiting some time after the initial signal change to make sure the signal value stabilizes. In this example, switch debouncing is implemented in a state model. The idea is to introduce a *confirmation* interval. After the signal transition from inactive to active is reported (*i.e.* by the pin interrupt), a timer is used to measure the confirmation interval. During the confirmation interval, the input signal is ignored. At the end of the confirmation interval, the signal is read to determine if it is active. If it is active, the button was held down long enough for the signal to stabilize. If the signal is inactive at the end of the confirmation interval, it is deemed a "glitch" and the original notification of the change in pin state is ignored. The time needed for the signal to stabilize is quite short, usually much less than few milliseconds. However, a long confirmation interval can be used to *slow* the recognition of the button press. This would force a user to hold down the button longer before the press would be recognized and can be used to prevent an accidental, glancing touch of a button from triggering a response.

Another useful concept is one of a *relaxation* interval. The *relaxation* interval is an amount of time to wait after a button press is confirmed but before arming the pin for another press. The relaxation interval has the effect of limiting the rate at which the button can be pressed and still be recognized.

The following state model diagram shows these ideas.

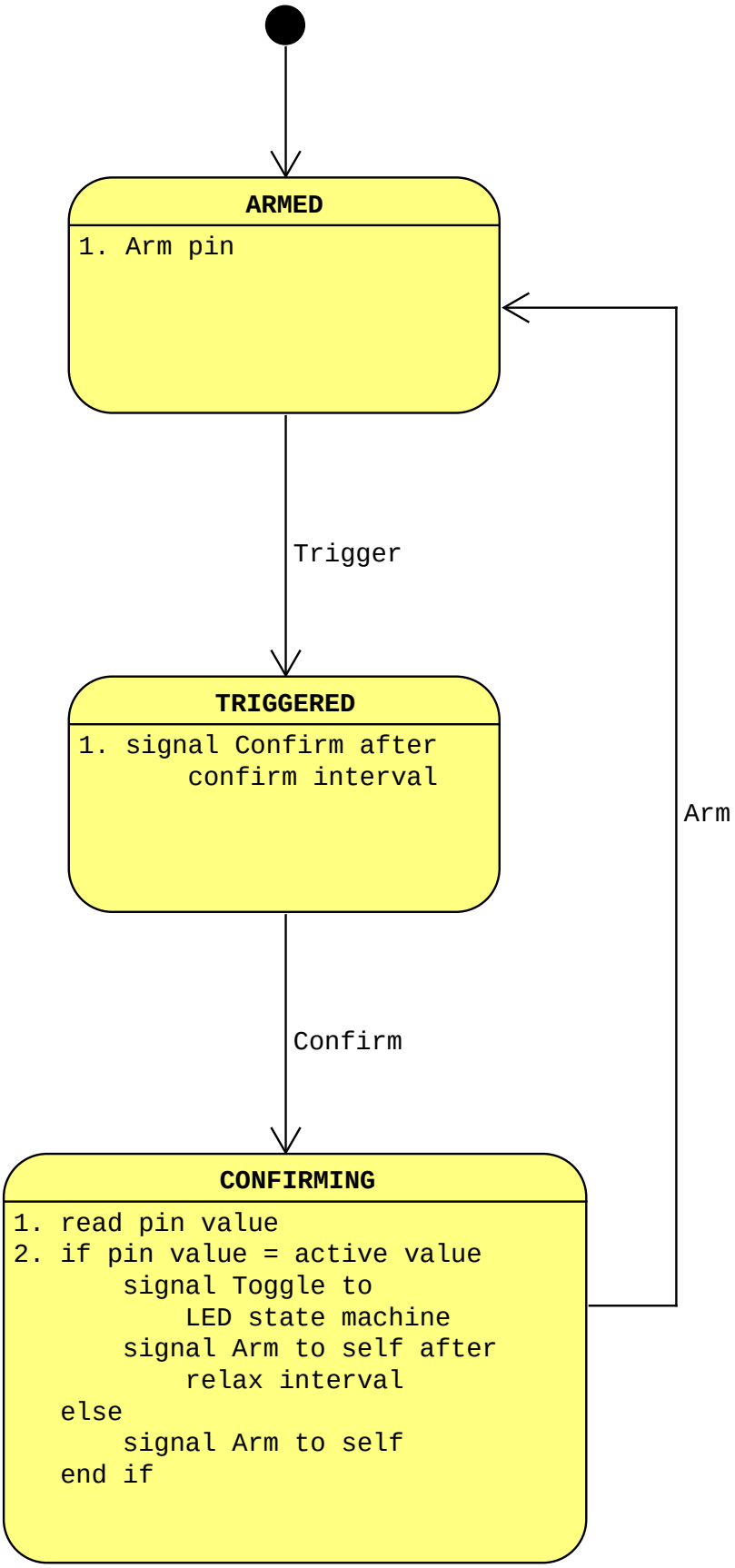

Figure 8.8: Bouncy State Model

The pattern for defining the Bouncy state model is the same as was used above for defining the Blinky state model. The definition is shown with no additional explanation.

```
<<button blinky: state model>>=
typedef enum {
    buts_ARMED = 0,
    buts_TRIGGERED,
    buts_CONFIRMING,

    buts_STATE_COUNT    // last
} Button_State ;
```

```
<<button blinky: state model>>=
static const
SSM_DEFINE_STATE_NAMES(bouncy_model, buts_STATE_COUNT) = {
    SSM_NAME_STATE(buts_ARMED, ARMED),
    SSM_NAME_STATE(buts_TRIGGERED, TRIGGERED),
    SSM_NAME_STATE(buts_CONFIRMING, CONFIRMING),
} ;
```

```
<<button blinky: state model>>=
typedef enum {
    bute_Trigger = 0,
    bute_Confirm,
    bute_Arm,

    bute_EVENT_COUNT    // last
} Button_Event ;
```

```
<<button blinky: state model>>=
static const
SSM_DEFINE_EVENT_NAMES(bouncy_model, bute_EVENT_COUNT) = {
    SSM_NAME_EVENT(bute_Trigger, Trigger),
    SSM_NAME_EVENT(bute_Confirm, Confirm),
    SSM_NAME_EVENT(bute_Arm, Arm),
} ;
```

```
<<button blinky: state model>>=
static const
SSM_DEFINE_ACTIVITY_TABLE(bouncy_model, buts_STATE_COUNT) = {
    SSM_DEFINE_ACTIVITY(buts_ARMED, armed_activity),
    SSM_DEFINE_ACTIVITY(buts_TRIGGERED, triggered_activity),
    SSM_DEFINE_ACTIVITY(buts_CONFIRMING, confirming_activity),
} ;
```

```
<<button blinky: forward references>>=
static void armed_activity(SSM_Machine *const self) ;
static void triggered_activity(SSM_Machine *const self) ;
static void confirming_activity(SSM_Machine *const self) ;
```

Again, it is most important to supply a completed transition matrix.

Table 8.2: Button Debounce Transition Matrix

	Trigger	**Confirm**	**Arm**
ARMED	TRIGGERED	CH	CH
TRIGGERED	CH	CONFIRMING	CH
CONFIRMING	CH	CH	ARMED

```
<<button blinky: state model>>=
static const
SSM_DEFINE_TRANS_TABLE(bouncy_model, buts_STATE_COUNT, bute_EVENT_COUNT) = {
    SSM_DEFINE_TRANSITION(buts_ARMED, bute_Trigger, buts_TRIGGERED),
    SSM_DEFINE_TRANSITION(buts_ARMED, bute_Confirm, SSM_State_CH),
    SSM_DEFINE_TRANSITION(buts_ARMED, bute_Arm, SSM_State_CH),

    SSM_DEFINE_TRANSITION(buts_TRIGGERED, bute_Trigger, SSM_State_CH),
    SSM_DEFINE_TRANSITION(buts_TRIGGERED, bute_Confirm, buts_CONFIRMING),
    SSM_DEFINE_TRANSITION(buts_TRIGGERED, bute_Arm, SSM_State_CH),

    SSM_DEFINE_TRANSITION(buts_CONFIRMING, bute_Trigger, SSM_State_CH),
    SSM_DEFINE_TRANSITION(buts_CONFIRMING, bute_Confirm, SSM_State_CH),
    SSM_DEFINE_TRANSITION(buts_CONFIRMING, bute_Arm, buts_ARMED),
} ;
```

Bouncy State Machine

Like the Blinky state machine, a structure is defined to hold the state machine dispatch mechanisms and the parametric data needed by the state actions.

```
<<button blinky: data type declarations>>=
typedef struct {
    SSM_Machine machine ;
    unsigned pin ;
    unsigned pin_active_value ;
    unsigned confirm_interval_ms ;
    unsigned relax_interval_ms ;
    Blinky_Machine *const controlled_led ;
} Bouncy_Machine ;
```

machine
: The state machine that controls an individual button. Note, the machine is *contained* within the parametric data structure and not access by reference. This member holds the current state of the machine and references to the state model which contains the transition function.

pin
: The GPIO pin number attached to the button.

pin_active_value
: The value of the pin input which is considered as active. Valid values are 0 or 1.

confirm_interval_ms
: The number of milliseconds to wait for the button input signal to stabilize.

relax_interval_ms
: The number of milliseconds to wait after a recognized button press before rearming the button for subsequent presses.

controlled_led
: A pointer to the state machine for the LED which is to be controlled by the button.

The state activity functions use the same `CONTAINER_OF` macro to gain access to the parametric data associated with the state machine.

```
<<button blinky: state model>>=
static void
```

```
armed_activity(
    SSM_Machine *const self)
{
    Bouncy_Machine *bm = CONTAINER_OF(self, Bouncy_Machine, machine) ;

    int status = dev_gpio_arm(bm->pin) ;
    assert(status == 0) ; (void)status ;
}
```

```
<<button blinky: state model>>=
static void
triggered_activity(
    SSM_Machine *const self)
{
    Bouncy_Machine *bm = CONTAINER_OF(self, Bouncy_Machine, machine) ;

    TIMQ_ElementID timq_elem = dev_timq_insert(0,
            dev_util_timq_ms_to_ticks(bm->confirm_interval_ms), 0,
            (SVC_DevNotifyClosure)ds_bouncy_Confirm) ;              // ❶
    assert(timq_elem >= 0) ; (void)timq_elem ;
}
```

❶ Note that since the delay times are not cancelled, there is no need to save the timer element ID.

```
<<button blinky: state model>>=
static void
confirming_activity(
    SSM_Machine *const self)
{
    Bouncy_Machine *bm = CONTAINER_OF(self, Bouncy_Machine, machine) ;

    unsigned pin_value ;
    int status = dev_gpio_read(bm->pin, &pin_value) ;
    assert(status == 0) ; (void)status ;
    if (pin_value == bm->pin_active_value) {
        SSM_signal(&bm->controlled_led->machine, lede_Toggle) ;
        TIMQ_ElementID timq_elem = dev_timq_insert(0,
                dev_util_timq_ms_to_ticks(bm->relax_interval_ms), 0,
                (SVC_DevNotifyClosure)ds_bouncy_Arm) ;
        assert(timq_elem >= 0) ; (void)timq_elem ;
    } else {
        status = SSM_signalSelf(self, bute_Arm) ;
        assert(status) ;
    }
}
```

```
<<button blinky: state model>>=
static const
SSM_DEFINE_STATE_MODEL(bouncy_model, buts_ARMED,
        buts_STATE_COUNT, bute_EVENT_COUNT) ;
```

```
<<button blinky: constants>>=
#define BUTTON_PIN      27U
```

A state machine to detect button presses can now be defined. It signals each button press to control a blinky state machine. For the example, there is one state machine for the button and one for the LED. There could be as many button/LED state machine pairs as required.

```
<<button blinky: state model>>=
static Bouncy_Machine bouncy = {
    SSM_MACHINE_INITIALIZER(bouncy, machine, &bouncy_model, buts_ARMED,
            &button_blinky_context),
    .pin = BUTTON_PIN,
    .pin_active_value = 0,
    .confirm_interval_ms = 20U,
    .relax_interval_ms = 300U,
    .controlled_led = &blinky,
} ;
```

Background Notification Handling

The state activities of the state models contain code to make direct background requests for GPIO pin service and for Timer Queue service[3]. But when the input pin changes state or the timer queue element has expired, a background notification is queued. The background notifications must be mapped onto state machine events. This mapping completes the "circuit" between foreground and background processing.

For both the GPIO and timer cases, the ability of the background notifications to carry a piece of *closure* data is used to serve as the mechanism for mapping the notifications to state machine events. The GPIO and timer queue services are general services which are not concerned about what the ultimate use is. They simply report that a pin has changed state or an interval of time has expired. It is up to the code to map those basic concepts onto the state machines which drive the execution.

The push button is the simpler situation. Each input pin has a background proxy function associated with it and the desired event number to signal is passed as the closure value. In this case the event is symbolically encoded as, `bute_Toggle`. The following code shows how the pin attached to the button is configured for input.

```
<<button blinky: button configuration>>=
status = dev_gpio_in_config(bouncy.pin, gpin_intr_high_low,
        true, pad_pullup_none, button_notified,
        (SVC_DevNotifyClosure)bute_Trigger) ;
assert(status == 0) ;
```

Note the button pin is configured to disarm itself when it triggers. This allows us to ignore any glitch in the signal without a race condition.

The following code is executed when the background notification from the button pin is dispatched.

```
<<button blinky: state model>>=
static void
button_notified(
    SVC_DevGpioNotification const *const notification)
{
    Button_Event event = (Button_Event)notification->notify_closure ;
    assert(event < bute_EVENT_COUNT) ;
    SSM_signalAndRun(&bouncy.machine, event) ;
}
```

For the timer queue, the situation is different. The timer queue provides a single background proxy function for all the elements in the queue.

```
<<button blinky: timer configuration>>=
status = dev_timq_open(0, time_expired) ;
assert(status == 0) ;
```

However, each element carries its own closure data. The strategy is to create an encoding for the closure data which can be used to map a given timer queue element expiration to a state machine and an event for that machine. There are four possible uses of timing elements in the two state machines. They are encoded as four distinct delayed signals.

[3] This is not the best design, but simplifies the situation for the benefit of the example.

```
<<button blinky: constants>>=
typedef enum {
    ds_bouncy_Confirm = 0,
    ds_bouncy_Arm,
    ds_blinky_Turn_on,
    ds_blinky_Turn_off,
} Delayed_Signal ;
```

Each time a state activity invokes `dev_timq_insert`, it provides a closure value from among the Delayed_Signal enumerators. When the inserted timer queue element expires, the following function is dispatched as the background proxy for the notification.

```
<<button blinky: state model>>=
static void
time_expired(
    SVC_DevTimqNotification const *const notification)
{
    static struct {
        SSM_Machine *target ;
        unsigned event ;
    } const ds_map[] = {
        [ds_bouncy_Confirm] = {
            .target = &bouncy.machine,
            .event = (unsigned)bute_Confirm,
        },
        [ds_bouncy_Arm] = {
            .target = &bouncy.machine,
            .event = (unsigned)bute_Arm,
        },
        [ds_blinky_Turn_on] = {
            .target = &blinky.machine,
            .event = (unsigned)lede_Turn_on,
        },
        [ds_blinky_Turn_off] = {
            .target = &blinky.machine,
            .event = (unsigned)lede_Turn_off,
        },
    } ;

    Delayed_Signal signal = (Delayed_Signal)notification->notify_closure ;
    rtcheck_max(signal, COUNTOF(ds_map)) ;

    assert(ds_map[signal].target != NULL) ;

    SSM_signalAndRun(ds_map[signal].target, ds_map[signal].event) ;
}
```

The mapping of delayed signal enumerator value to state machine / event pairs is implemented as an array, indexed by the delayed signal value. Each element of the mapping array is a state machine / event pair which is signaled to the target state machine as an event. The net effect is, as it appears to the state machines, is an ordinary event. The effect as seen from a system perspective is that the signaled event was delayed in time from the execution of the state action.

Note that in both of the background proxy functions, the state machines are signaled events using the `SSM_signalAndRun` function. This function both signals the event and then immediately dispatches the event as well as any other events signaled as the consequence of the delivery of the original event. For example, when the button press fails to be confirmed, the **CONFIRMING** state signals the **Arm** event to itself in order to be ready for any subsequent button presses. If that is the path taken by the code of the **CONFIRMING** state activity, then the Bouncy state model ends up in the **ARMED** state by the time `SSM_signalAndRun` returns.

Button Blinky Main

The `main` function of the program follows our typical pattern. After initialization, an infinite event loop is entered.

```
<<button blinky: main>>=
void
main(void)
{
    int status = dev_gpio_out_config(blinky.pin, gpout_pushpull,
            pad_drive_12mA, pad_pullup_none) ;
    assert(status == 0) ; (void)status ;
    status = dev_gpio_write(blinky.pin, blinky.pin_inactive_op) ;

    <<button blinky: button configuration>>
    <<button blinky: timer configuration>>

    SSM_logEnable(&blinky.machine, true) ;          // ❶
    SSM_logEnable(&bouncy.machine, true) ;

    bool volatile forever ;
    sys_ctrl_busy_wait(&forever) ;
}
```

❶ The state machines can show an execution trace of the event dispatches. Examples of this are shown below.

Code Layout

```
<<tyranny-of-the-pins-button-blinky-test.c>>=
<<edit warning>>
<<copyright info>>

#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include "rtcheck.h"
#include "useful.h"
#include "sys_svc_req.h"
#include "dev_svc_timq_req.h"
#include "dev_svc_gpio_req.h"

<<button blinky: include files>>
<<button blinky: constants>>
<<button blinky: forward references>>
<<button blinky: data type declarations>>
<<button blinky: state model>>
<<button blinky: main>>
```

Tracing execution

The following is a trace of the event dispatch for a run of the example program. The trace shows the button press to start the LED blinking and a short time later a button press to stop the blinking. The trace entries give the name of the state machine and the event it receives. After the colon, the transition from current state to new state is given.

Event Dispatch Trace

```
bouncy-machine <- Trigger: ARMED -> TRIGGERED
bouncy-machine <- Confirm: TRIGGERED -> CONFIRMING
blinky-machine <- Toggle: IDLE -> ON
```

```
blinky-machine <- Turn_off: ON -> OFF
bouncy-machine <- Arm: CONFIRMING -> ARMED
blinky-machine <- Turn_on: OFF -> ON
blinky-machine <- Turn_off: ON -> OFF
blinky-machine <- Turn_on: OFF -> ON
blinky-machine <- Turn_off: ON -> OFF
blinky-machine <- Turn_on: OFF -> ON
blinky-machine <- Turn_off: ON -> OFF
bouncy-machine <- Trigger: ARMED -> TRIGGERED
bouncy-machine <- Confirm: TRIGGERED -> CONFIRMING
blinky-machine <- Toggle: OFF -> IDLE
bouncy-machine <- Arm: CONFIRMING -> ARMED
```

The first two lines show the background notification for the button press. The **bouncy-machine** transitions to **TRIGGERED** and then after the confirmation interval, it transitions to **CONFIRMING**. The button press as apparently confirmed since the **blinky-machine** received the **Toggle** event and turned itself on. Note that the LED is turned off before the GPIO pin for the button is rearmed. This is because the on interval for the LED is shorter than the relaxation interval for the button. After several on and off cycles of the LED, another button press is triggered and confirmed causing the LED blinking sequence to go idle.

Execution sequence diagram

A sequence diagram is another useful way to reason about the actions of the state machines and to show the collaboration of the state machines with device requests. The following diagram shows the details of the interactions between the Bouncy and Blinky state machines and the interactions between the state machines and the GPIO pins and timer elements. The diagrams shows, for this case, that most of the interactions are between the state machines and the GPIO device pins and the Timer Queue. Items 6 and 22 show the only interactions between the two state machines.

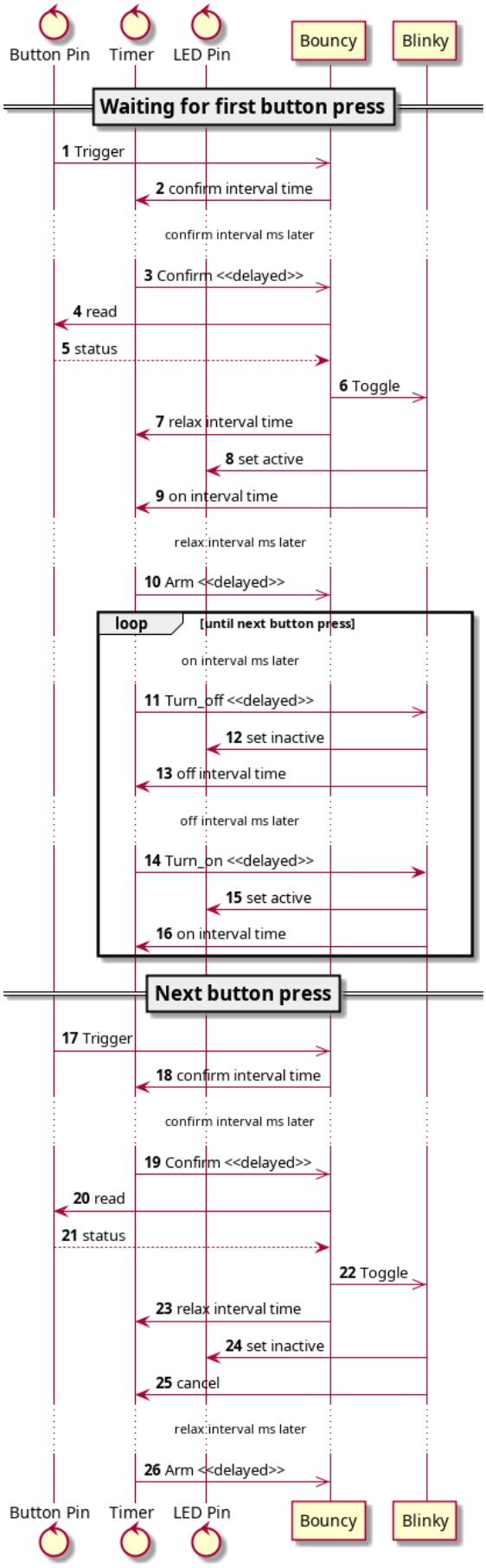

Figure 8.9: Sequence Diagram of Button Controlled LED Blinking

Evaluation

As mentioned at the beginning of this example, the manner in which background notifications are used to sequence background processing to include state machines was expanded. Previous examples had used direct, single level callbacks to initiate background processing in response to a background notification from an IRQ. It is fitting to examine the benefits and drawbacks of this method of controlling background execution.

Clearly, if the sole purpose of the application was to blink an LED under control of a button, the state machine formalism used is overly elaborate. There are simpler ways to accomplish manipulating some I/O pins. However, the following benefits can be seen.

- It is clear from the state model diagrams *what* computations take place without have to read through any code. The diagram does *not* indicate from where the events come, only what happens when they are received.
- There is direct association of state and action. If the state machine *transitions* into a state, the action code for that state is executed. This is true even if the state machines transitions back into the same state.
- State actions have access to parametric data which describes the characteristics of the object being controlled and provide any required data parameters.
- All transitions are accounted for. Any event defined for a state model is resolved by the transition matrix to some well defined action or non-transitioning situation.
- The same state model can be used for multiple button / LED combinations, *i.e.* the fixed cost of the state machine dispatch mechanism is amortized across all state machine instances.
- There is a clear separation of responsibility between the button and the LED state models. The button state model issues **Toggle** events to whatever state model it is associated to by its `controlled_led` data member. The LED state model is solely concerned with turning on and off the physical LED. The overall requirements to blink the LED under control of a button is achieved by *composing* the two state models where one state machine signals an event to another state machine. It is easier to reason about the behavior because of the separation than if the problem had been solved with a single state model which handled both the button debouncing and the timing of the LED illumination. A single state model solution would have involved more states and more paths through the state model.
- The interaction of the state machines can be shown in chronological order by examining the event dispatch trace. The execution trace becomes vital when a larger number of state machines interact.

The following drawbacks can be seen.

- It is not possible to read the source which implements the state models and determine the order in which processing happens. The state actions are placed in functions so it is easy to see what happens when a given state is entered. But execution sequencing is determined by generalized code operating over data structures and the same code is capable of dispatching events for any state model. The usual sequential execution model, *i.e.* one code statement is executed after another is lost in all but the state actions themselves.
- Breakpoint style debugging has limited utility. Outside of state actions, it is difficult to know which path a state machine will take. Since there may be multiple state machines executing according to the same state model, tracking instance specific execution is difficult within a breakpoint style debugging scheme.
- Specifying the state model using the pre-processor macros is laborious. For a few state models, one can simply roll up your sleeves and push through. However, if there were 30 or 40 state models, as might be found in a larger application, the pre-processor macros don't scale well to handle the amount of specification data required.
- Much of the specification information required for a state model amounts to arbitrary encoding of semantically meaningful names into integers. That encoding is somewhat fragile, despite our use of *designated initializers* to reduce the order dependency of generating initialized array variables for the state model specification data. When the inevitable changes to a state model arrive, the encoding must be carefully preserved. This is more difficult given the manner in which the state model is specified in this example, since adding or subtracting a state or event requires changes in multiple places.

It will come as no surprise that when background execution is discussed in later parts of this book, techniques will be employed that seek to maximize the benefits while minimizing the drawbacks.

Chapter 9

Speak To Me

Asynchronous serial communications was invented long before microcontrollers. Universal Asynchronous Receiver-Transmitter (UART) peripherals are commonly included in microcontrollers SOC's. They support the lowest common denominator for communications. With the advent of USB-to-UART bridge chips, UART peripherals are much more convenient to use, eliminating the need for intricate cable connections.

This chapter shows how asynchronous serial communication is implemented in the background/foreground scheme using the Apollo 3 UART module. UART's have a long and trying history of usage which has resulted in many variations of the communication schemes which use them. Serial communications by UART depends upon many external factors. For example, it is necessary for both sides in the communications to agree on speed, character size, parity inclusion, and character framing. This agreement must be done *a priori* and *out of band* [1]. The lack of any canonical link frame is another distinct problem. For example, an Ethernet packet is framed by the protocol so that the boundary of a packet can be determined as it is received. For UART communications, many such framings are in common use and for ordinary terminal I/O meant for a human, there are several conventions used to determine how lines (framing in this case) are terminated. This legacy of open-loop control creates substantial "accidental" complexity, but the simplicity and common availability of a UART warrants a workaround for its shortcomings. Again, no attempt is made to provide a comprehensive implementation which supports all the many variations. Only those configuration features of a UART that are most commonly used are implemented and those projects which have specific needs bear the burden of the additional work involved.

Design Overview

The UART code is designed to meet the following characteristics:

- Receiving and transmitting data is interrupt driven. The IRQ handler code will transfer data to and from the UART FIFO registers. The UART peripheral on the Apollo 3 does not support DMA operations, so the processor must move the data on and off the peripheral.
- Since there are several possible link frames formats, receive and transmit data are buffered in the foreground to allow for determining if a link frame has been received or to provide additional formatting for building a frame for transmission. Unlike other cases where buffering or memory management in foreground processing was avoided, this design uses a buffer on both the transmit and receive sides.
- Potential input frames are detected by a set of framing procedures. For transmitting, the TX framing adds bytes into the outgoing data to make up a frame. For receiving, the raw bytes received may have characters removed to match the type of the framing. Passing through all characters unmodified is also supported so that background processing can implement its own link framing conventions. The provided framing mechanism allows us to handle raw receive/transmit, terminal oriented I/O, and other framing schemes.
- Options to configure the UART device characteristics are limited. The baud rate and hardware flow control can be configured. The peripheral device is otherwise set to 8-bit characters, no parity, and one stop bit as these are common in current usage.

[1] It is sometimes possible to do automatic baud rate determination

The following figure shows a simplified schematic of how data is transferred to and from the UART.

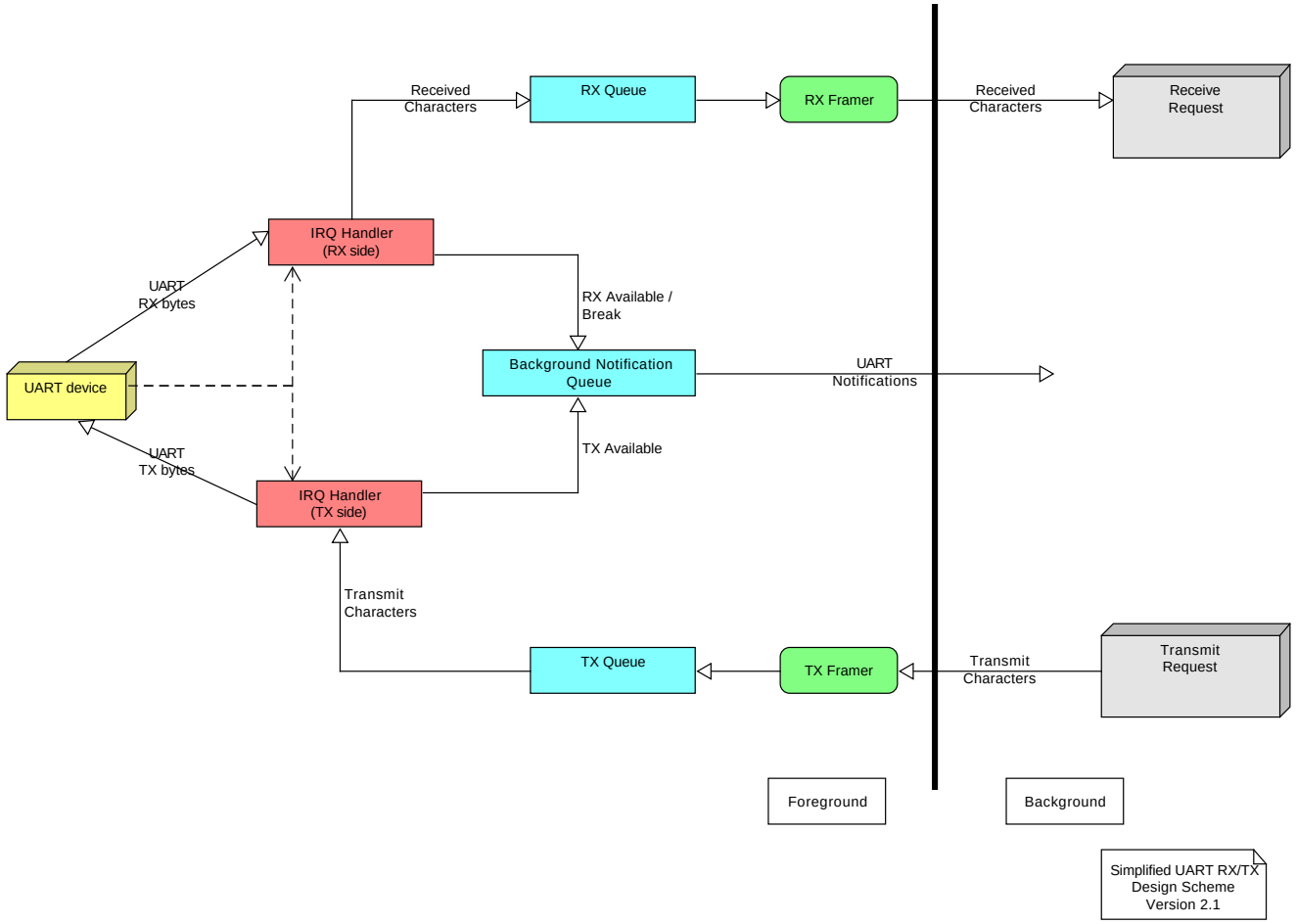

Figure 9.1: Simplified UART Device Design

Consider the receive path in the previous diagram. The foreground code has an internal queue it uses to hold the bytes as received from the UART. When the receive FIFO has reached its threshold, the UART device raises an interrupt and control transfers to the portion of the IRQ handler used for receiving. The handler reads bytes from the UART device FIFO and writes them to a receive queue. If the queue becomes full, then reception either overruns or, if hardware flow of control is supported, the transmitting peer is signaled that it must hold up transmission. When receive requests are made by the background, the RX framing removes characters from the RX queue to enforce link frame boundaries. Notifications indicate that additional characters are available and are posted when the previous receive request was not completely filled (*i.e.* a “short” read) and new characters have arrived.

Background processing is responsible for requesting incoming data until it receives what it expects and to use RX available notifications when more is expected.

The transmit path is similar. Background processing makes transmit requests to send data. The transmit data is placed in the TX queue by the TX framing. The UART is fed characters from the TX queue when it has space in its FIFO for more outgoing characters. Background notifications are issued when a transmit request was not completely fulfilled (*i.e.* a “short” write) and there is now space available in the transmit buffer.

There is also a concept of a session with the UART. Before sending or receiving data, the application must *open* the UART. Opening the UART provides the address of a background proxy function which receives the RX and TX notifications.

Applications should take care to only open the UART for an ongoing communications session and to close it when finished in order to conserve power. The objective is to open the device, promptly do what is required, and close the device immediately

when no longer needed. However, for some UART needs, *e.g.* console I/O used for status reporting and debugging, a UART may be expected to be left open for the lifetime of the program.

The following sections describe the details of how the UART requests are designed to operate and give the "C" source code.

Representing a UART

The following data structure contains the elements needed to control communications using the UART peripheral and implement the buffer queuing.

UART_ControlBlock

```
<<dev realm uart proxy: uart control block type declaration>>=
typedef struct uart_controlblock {
    UART_TxControl tx_control ;
    UART_RxControl rx_control ;
    UART_Access const *const access ;
    SVC_DevUartNotifyProxy proxy ;
    SVC_DevNotifyClosure closure ;
} UART_ControlBlock ;
```

tx_control
: The data required control the transmit side of the UART.

rx_control
: The data required to control the receive side of the UART.

access
: A pointer to data used to access the registers for the UART and any GPIO pin information for pins used by the UART. This information does not not change at run time.

proxy
: A pointer to a background notification proxy function used to report I/O buffer status. Its value is set when the device is opened.

closure
: A data value supplied by the user of the UART device and returned in the background notifications from the UART.

UART TX and RX Controls

Since the UART device is capable of *full duplex* communications, the transmit and receive sides are treated separately. The data required to handle both sides is similar, but not identical. For transmission, the following data structure is used.

UART_TxControl

```
<<dev realm uart proxy: uart tx control type declaration>>=
typedef struct {
    UART_TxQueue tx_queue ;
    UART_TxFramer tx_framer ;
    uint8_t tx_set_aside[sizeof(uint32_t)] ;
    size_t tx_itor ;
    bool tx_short_write ;
} UART_TxControl ;
```

tx_queue
A FIFO queue to support buffering for the transmit side.

tx_framer
A pointer to the transmit framing function.

tx_set_aside
Character storage available to the transmit framing function for any holding state history of the previously seen characters.

tx_itor
An index variable used as an iterator in the TX queue. This member is available to those framing functions which require tracking individual characters in the TX queue.

tx_short_write
A boolean status indicating whether the last transmit request was able to queue all the requested bytes.

For receiving, there are differences in the receive queue and receive framing functions. The following data type records the information for controlling the receive side.

UART_RxControl

```
<<dev realm uart proxy: uart rx control type declaration>>=
typedef struct {
    UART_RxQueue rx_queue ;
    UART_RxFramer rx_framer ;
    uint8_t rx_set_aside[sizeof(uint32_t)] ;
    size_t rx_itor ;
    bool rx_short_read ;
} UART_RxControl ;
```

rx_queue
A FIFO queue to support buffering for the receive side.

rx_framer
A pointer to a receive framing function.

rx_set_aside
Character storage available to the receive framing function for any holding state history of the previously seen characters.

rx_itor
An index variable used as an iterator in the RX queue. This member is available to those framing functions which require tracking individual characters in the RX queue.

rx_short_read
A boolean status indicating whether fewer than the number of bytes requested in the last receive request were transferred.

I/O Queue

The I/O queues used for the UART are simple circular queues held in an array. For transmitting, the queue data structure is given as follows.

```
<<dev realm uart proxy: io queue type declaration>>=
typedef struct {
    size_t head ;
    size_t tail ;
    uint8_t storage[DEV_UART_TX_QUEUE_SIZE] ;
} UART_TxQueue ;
```

The queue is operated using indices rather than pointers. The index design was chosen to make the modulus arithmetic that is required by the TX and RX framing functions easier.

The RX queue is slightly different. The UART peripheral device augments each received byte with 4 bits of error status. The error status is tracked through the RX queue and error information is handed back whenever the corresponding received byte is read by background processing. So, 16-bit quantities are required to hold the RX data extended by its status information.

```
<<dev realm uart proxy: io queue type declaration>>=
typedef struct {
    size_t head ;
    size_t tail ;
    uint16_t storage[DEV_UART_RX_QUEUE_SIZE] ;
} UART_RxQueue ;
```

The queue storage is sized generously enough for common usage. Much more transmitted data is expected than received data.

```
<<dev realm uart proxy: constants>>=
#ifndef DEV_UART_RX_QUEUE_SIZE
#   define DEV_UART_RX_QUEUE_SIZE   256U
#endif /* DEV_UART_RX_QUEUE_SIZE */

#ifndef DEV_UART_TX_QUEUE_SIZE
#   define DEV_UART_TX_QUEUE_SIZE   2048U
#endif /* DEV_UART_TX_QUEUE_SIZE */
```

The index calculations for the queues must be done with modulus arithmetic. A couple of inline functions save some typing. Also note that by using a compile time known constant as the modulus, the compile may be able to optimize away a potential division.

```
<<dev realm uart proxy: static inline functions>>=
static inline size_t
modulo_rx_queue(
    size_t index)
{
    return index % DEV_UART_RX_QUEUE_SIZE ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline size_t
modulo_tx_queue(
    size_t index)
{
    return index % DEV_UART_TX_QUEUE_SIZE ;
}
```

I/O Queue Operations

The following are a set of operations on the I/O queue. Because of the difference between the TX queue and RX queue data types, the `_Generic` construct is used to provide some rudimentary function overloading.

ioqueue_init

Initializes the IO_Queue, `queue`, to be empty.

```
<<dev realm uart proxy: macros>>=
#define ioqueue_init(queue)     _Generic((queue),   \
    UART_TxQueue *: txqueue_init,          \
    UART_RxQueue *: rxqueue_init)(queue)
```

```
<<dev realm uart proxy: static inline functions>>=
static inline void
txqueue_init(
    UART_TxQueue *const queue)
{
    queue->head = queue->tail = 0 ;
    memset(queue->storage, 0, sizeof(queue->storage)) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline void
rxqueue_init(
    UART_RxQueue *const queue)
{
    queue->head = queue->tail = 0 ;
    memset(queue->storage, 0, sizeof(queue->storage)) ;
}
```

ioqueue_empty

Returns `true` if `queue` is empty and `false` otherwise.

```
<<dev realm uart proxy: macros>>=
#define ioqueue_empty(queue)    _Generic((queue),   \
    UART_TxQueue *: txqueue_empty,                \
    UART_RxQueue *: rxqueue_empty)(queue)
```

```
<<dev realm uart proxy: static inline functions>>=
static inline bool
txqueue_empty(
    UART_TxQueue *const queue)
{
    return queue->head == queue->tail ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline bool
rxqueue_empty(
    UART_RxQueue *const queue)
{
    return queue->head == queue->tail ;
}
```

ioqueue_full

Returns `true` if `queue` is full and `false` otherwise.

```
<<dev realm uart proxy: macros>>=
#define ioqueue_full(queue)     _Generic((queue),   \
    UART_TxQueue *: txqueue_full,             \
    UART_RxQueue *: rxqueue_full)(queue)
```

```
<<dev realm uart proxy: static inline functions>>=
static inline bool
txqueue_full(
    UART_TxQueue *const queue)
{
    return modulo_tx_queue(queue->tail + 1) == queue->head ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline bool
rxqueue_full(
    UART_RxQueue *const queue)
{
    return modulo_rx_queue(queue->tail + 1) == queue->head ;
}
```

ioqueue_count

Returns the number of items in `queue`.

```
<<dev realm uart proxy: macros>>=
#define ioqueue_count(queue)     _Generic((queue),  \
    UART_TxQueue *: txqueue_count,                \
    UART_RxQueue *: rxqueue_count)(queue)
```

```
<<dev realm uart proxy: static inline functions>>=
static inline size_t
txqueue_count(
    UART_TxQueue *const queue)
{
    return modulo_tx_queue(queue->tail - queue->head) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline size_t
rxqueue_count(
    UART_RxQueue *const queue)
{
    return modulo_rx_queue(queue->tail - queue->head) ;
}
```

ioqueue_insert

Inserts `item` into `queue`. Returns `true` if the insertion succeeded and `false` if `queue` was full.

```
<<dev realm uart proxy: macros>>=
#define ioqueue_insert(queue, item)     _Generic((queue),   \
    UART_TxQueue *: txqueue_insert,                        \
    UART_RxQueue *: rxqueue_insert)(queue, item)
```

```
<<dev realm uart proxy: static function definitions>>=
static bool
txqueue_insert(
    UART_TxQueue *const queue,
    uint8_t item)
{
    bool inserted = false ;

    size_t next_tail = modulo_tx_queue(queue->tail + 1) ;    // ❶
    if (next_tail != queue->head) {
        queue->storage[queue->tail] = item ;
        queue->tail = next_tail ;
        inserted = true ;
    }

    return inserted ;
}
```

❶ One storage slot is used to detect overflow. If insertion would cause the tail pointer to equal the head pointer then the queue is full and the insertion attempt fails because of overflow.

```
<<dev realm uart proxy: static inline functions>>=
static bool
rxqueue_insert(
    UART_RxQueue *const queue,
    uint16_t item)
{
    bool inserted = false ;

    size_t next_tail = modulo_rx_queue(queue->tail + 1) ;
    if (next_tail != queue->head) {
        queue->storage[queue->tail] = item ;
        queue->tail = next_tail ;
        inserted = true ;
    }

    return inserted ;
}
```

ioqueue_remove

Removes an item from `queue` and returns the item via the pointer reference, `item_ref`. The function returns `true` if an item was removed and `false` if `queue` was empty.

```
<<dev realm uart proxy: macros>>=
#define ioqueue_remove(queue, item_ref)     _Generic((queue),    \
    UART_TxQueue *: txqueue_remove,                              \
    UART_RxQueue *: rxqueue_remove)(queue, item_ref)
```

```
<<dev realm uart proxy: static function definitions>>=
static bool
txqueue_remove(
    UART_TxQueue *const queue,
    uint8_t *tx_item_ref)
{
    bool removed = false ;

    if (!ioqueue_empty(queue)) {
```

```
        *tx_item_ref = queue->storage[queue->head] ;
        queue->head = modulo_tx_queue(queue->head + 1) ;
        removed = true ;
    }

    return removed ;
}
```

```
<<dev realm uart proxy: static function definitions>>=
static bool
rxqueue_remove(
    UART_RxQueue *const queue,
    uint16_t *rx_item_ref)
{
    bool removed = false ;

    if (!ioqueue_empty(queue)) {
        *rx_item_ref = queue->storage[queue->head] ;
        queue->head = modulo_rx_queue(queue->head + 1) ;
        removed = true ;
    }

    return removed ;
}
```

Framing Transformations

When transmit data is copied into the TX queue and when receive data is copied out of the RX queue, the data stream is framed by adding (in the TX case) or subtracting (in the RX case) bytes to either encapsulate or decode the data. This design supports a general mechanism to invoke framing functions and supplies three types of commonly used framing:

Identity
: Data is passed through unmodified.

Terminal
: When receiving, carriage return (CR), linefeed (LF) or CR/LF sequences are converted to LF and the frame is returned after any LF character. When transmitting, LF is converted to CR/LF. This allows input records to be LF separated internally, but converted to CR/LF sequences on output.

Interactive
: When transmitting, interactive framing is the same as terminal framing. When receiving, interactive framing echos input characters and responds to line editing characters (*e.g.* backspace (BS)). Interactive framing is suitable for use as a console directly interfacing to a human.

```
<<dev realm uart param: data type declarations>>=
typedef enum {
    uart_frame_identity,
    uart_frame_terminal,
    uart_frame_interactive,

    uart_frame_type_count        // last
} UART_FramingType ;
```

The framing mechanism can support an arbitrary number of framing transformers. The three provided are for common usage, but framing functions for better defined link frames such as Serial Line Internet Protocol (SLIP) or common modem communications frames can be integrated into the mechanism.

Because the TX and RX framing functions transfer data from and to unprivileged memory, these functions are required to perform that transfer in an unprivileged manner.

There are separate framing functions for transmit and receive.

```
<<dev realm uart proxy: framer type declarations>>=
typedef size_t (*UART_TxFramer)(struct uart_controlblock *const ucb,
        size_t len, uint8_t const tx_buf[len]) ;
```

```
<<dev realm uart proxy: framer type declarations>>=
typedef size_t (*UART_RxFramer)(struct uart_controlblock *const ucb,
        size_t len, uint8_t rx_buf[len], UART_RxErrStatus *const status_ref) ;
```

Identity Framing Transformation

The identity framer performs no transformation on characters and passes through all characters received or transmitted. This type of framing can be used by background processing for application specific framing.

```
<<dev realm uart proxy: forward references>>=
static size_t
ident_tx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t const tx_buf[len]) ;
```

For transmitting, the identify framer just copies the data from the caller's buffer into the TX buffer. The copy continues until there is either no more caller data or the TX buffer is full

```
<<dev realm uart proxy: static function definitions>>=
static size_t
ident_tx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t const tx_buf[len])
{
    assert(ucb != NULL) ;

    UART_TxQueue *const tx_queue = &ucb->tx_control.tx_queue ;
    size_t count = 0 ;
    for ( ; count < len && !ioqueue_full(tx_queue) ; count++) {
        uint8_t tx_char = READ_UNPRIV(&tx_buf[count]) ;
        bool inserted = ioqueue_insert(tx_queue, tx_char) ;
        assert(inserted) ; (void)inserted ;
    }

    return count ;
}
```

```
<<dev realm uart proxy: forward references>>=
static size_t
ident_rx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t rx_buf[len],
    UART_RxErrStatus *const status_ref) ;
```

Similar to transmit, the identity framer for receiving copies data from the RX buffer to the caller's buffer. Because receiving can detect errors, each character is checked for error and any error stops the transfer. With no error, the transfer stops when there is no more data in the TX buffer or the caller's buffer space is filled.

```
<<dev realm uart proxy: static function definitions>>=
static size_t
ident_rx_framer(
```

```
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t rx_buf[len],
    UART_RxErrStatus *const status_ref)
{
    assert(ucb != NULL) ;
    assert(status_ref != NULL) ;

    UART_RxQueue *const rx_queue = &ucb->rx_control.rx_queue ;
    *status_ref = uart_success ;

    uint16_t rx_xfer_item ;
    size_t count = 0 ;
    for ( ; *status_ref == uart_success && count < len &&
            ioqueue_remove(rx_queue, &rx_xfer_item) ; count++) {
        WRITE_UNPRIV((uint8_t)rx_xfer_item, &rx_buf[count]) ;
        *status_ref = (UART_RxErrStatus)btwd_bits_extract(
                rx_xfer_item, RX_ERR_FIELD_WIDTH, RX_ERR_FIELD_OFFSET) ;
    }

    return count ;
}
```

The error status resides in the four bits 8 - 11. Symbolic constants define the RX error bit fields.

```
<<dev realm uart proxy: constants>>=
#define RX_ERR_FIELD_WIDTH      4
#define RX_ERR_FIELD_OFFSET     8
```

Terminal I/O Framing Transformation

Terminal I/O framing is intended for use with conventional CR/LF terminated records. The framing is flexible to allow the usual permutations of CR and LF as either solitary record terminators or as a sequence to indicate frame termination.

```
<<dev realm uart proxy: forward references>>=
static size_t
terminal_tx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t const tx_buf[len]) ;
```

The implementation of `terminal_tx_framer` uses the `tx_set_aside` buffer to hold the last character seen in the stream. This gives us the necessary state information to handle inserting a CR/LF sequence into the transmit buffer when either a single LF character or a CR/LF sequence is present. Note that isolated CR characters are simply passed through.

```
<<dev realm uart proxy: static function definitions>>=
static size_t
terminal_tx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t const tx_buf[len])
{
    assert(ucb != NULL) ;

    UART_TxControl *const tx_control = &ucb->tx_control ;
    UART_TxQueue *const tx_queue = &tx_control->tx_queue ;

    size_t count = 0 ;
    while (count < len && !ioqueue_full(tx_queue)) {
        uint8_t tx_char = READ_UNPRIV(&tx_buf[count]) ;
```

```
        if (tx_char == '\n' && tx_control->tx_set_aside[0] != '\r') {
            tx_control->tx_set_aside[0] = '\r' ;
            bool inserted = ioqueue_insert(tx_queue, '\r') ;
            assert(inserted) ; (void)inserted ;
            continue ;                                          // ❶
        }

        tx_control->tx_set_aside[0] = tx_char ;
        bool inserted = ioqueue_insert(tx_queue, tx_char) ;
        assert(inserted) ; (void)inserted ;
        count++ ;
    }

    return count ;
}
```

❶ The use of the `continue` statement may seem unusual as it forces the NL character to be transferred from the caller supplied buffer twice. However, this scheme works should the I/O queue become full after inserting the CR character, *i.e.* in the loop body, at least one character can be inserted, and the `continue` forces a re-evaluation of the loop control to insure there is a place for the NL.

```
<<dev realm uart proxy: forward references>>=
static size_t
terminal_rx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t rx_buf[len],
    UART_RxErrStatus *const status_ref) ;
```

Similar to the TX framer, the `terminal_rx_framer` uses the `rx_set_aside` buffer to handle CR/LF sequences. The logic is to replace all CR characters with LF and to drop all LF characters immediately preceded by a CR.

```
<<dev realm uart proxy: static function definitions>>=
static size_t
terminal_rx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t rx_buf[len],
    UART_RxErrStatus *const status_ref)
{
    assert(ucb != NULL) ;
    assert(status_ref != NULL) ;

    UART_RxControl *const rx_control = &ucb->rx_control ;
    UART_RxQueue *const rx_queue = &rx_control->rx_queue ;

    uint16_t rx_xfer_item ;
    *status_ref = uart_success ;
    size_t count = 0 ;
    while (*status_ref == uart_success &&
            count < len && ioqueue_remove(rx_queue, &rx_xfer_item)) {
        uint8_t rx_char = (uint8_t)rx_xfer_item ;

        if (rx_char == '\n' && ucb->rx_control.rx_set_aside[0] == '\r') {
            rx_control->rx_set_aside[0] = '\n' ;
            continue ;
        } else if (rx_char == '\r') {
            rx_control->rx_set_aside[0] = '\r' ;
            rx_char = '\n' ;
```

```
        } else {
            rx_control->rx_set_aside[0] = rx_char ;
        }

        WRITE_UNPRIV(rx_char, &rx_buf[count]) ;
        count++ ;

        *status_ref = (UART_RxErrStatus)btwd_bits_extract(rx_xfer_item,
                RX_ERR_FIELD_WIDTH, RX_ERR_FIELD_OFFSET) ;
    }

    return count ;
}
```

Interactive Framing Transformation

The interactive framing functions are intended for using the UART directly with a human. Human console interactions are record oriented, but need to support echoing of input characters and line editing. It operates by holding a line in reserve in the buffer so that backspace and other line editing can occur. Once the line is terminated with a CR, it is made available to the background. There is nothing special about interactive output: it is the same as terminal output.

```
<<dev realm uart proxy: forward references>>=
static size_t
interact_rx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t rx_buf[len],
    UART_RxErrStatus *const status_ref) ;
```

For interactive receive framing, an entire line of input is held in the RX buffer so that line editing commands can be performed. This framer uses the `rx_itor` member of the receive controls to store the position in the RX buffer where it has last examined the input. The first task performed is to determine if there is a complete frame in the RX buffer. If there is, then the frame transferred to the caller's buffer.

```
<<dev realm uart proxy: static function definitions>>=
static size_t
interact_rx_framer(
    UART_ControlBlock *const ucb,
    size_t len,
    uint8_t rx_buf[len],
    UART_RxErrStatus *const status_ref)
{
    assert(ucb != NULL) ;
    assert(status_ref != NULL) ;

    *status_ref = uart_success ;
    size_t count = 0 ;

    if (scan_for_frame(ucb)) {
        UART_RxControl *const rx_control = &ucb->rx_control ;
        UART_RxQueue *const rx_queue = &rx_control->rx_queue ;
        size_t rx_itor = rx_control->rx_itor ;

        for ( ; count < len && rx_queue->head != rx_itor &&
                *status_ref == uart_success ;                                    // ❶
                rx_queue->head = modulo_rx_queue(rx_queue->head + 1)) {
            uint16_t rx_data  = rx_queue->storage[rx_queue->head] ;

            WRITE_UNPRIV((uint8_t)rx_data, &rx_buf[count]) ;
            count++ ;
```

```
            *status_ref = (UART_RxErrStatus)btwd_bits_extract(rx_data,
                    RX_ERR_FIELD_WIDTH, RX_ERR_FIELD_OFFSET) ;
        }
    }

    return count ;
}
```

❶ Note, only transfer up to the `rx_itor` position. This is as far as the characters in the RX buffer have been examined in determining that an interactive frame was found.

The `scan_for_frame` function performs the heavy lifting required to manage line editing within the RX buffer.

```
<<dev realm uart proxy: forward references>>=
static bool
scan_for_frame(
    UART_ControlBlock *const ucb) ;
```

This design only supports backspace (and delete) characters to remove the last input character and the `^U` character (kill) to discard the entire frame. Otherwise, frame detection is as expected, with CR (or a CR/LF sequence) ending the frame. Received characters are also echoed. This includes the necessary echo sequence to handle backspace and line kill, *i.e.* a sequence of backspaces and spaces are transmitted to erase the character from the interactive terminal display. Note that since the RX framer only executes when there is an ongoing attempt to read UART input, echo only occurs when background processing is executing a receive request.

```
<<dev realm uart proxy: static function definitions>>=
static bool
scan_for_frame(
    UART_ControlBlock *const ucb)
{
    assert(ucb != NULL) ;

    static uint8_t bs_echo[] = {'\b', ' ', '\b'} ;
    static uint8_t crlf_echo[] = {'\r', '\n'} ;

    UART_RxControl *const rx_control = &ucb->rx_control ;
    UART_RxQueue *const rx_queue = &rx_control->rx_queue ;
    uint16_t *rx_storage = rx_queue->storage ;
    UART0_Type *const uart = ucb->access->uart ;
    bool found_eoframe = false ;

    while (!(found_eoframe || rx_control->rx_itor == rx_queue->tail)) {
        uint8_t rx_char = (uint8_t)rx_storage[rx_control->rx_itor] ;

        if (rx_char == '\r') {
            put_tx_array(uart, sizeof(crlf_echo), crlf_echo) ;
            rx_queue->storage[rx_control->rx_itor] = '\n' ;
            rx_control->rx_itor = modulo_rx_queue(rx_control->rx_itor + 1) ;
            rx_control->rx_set_aside[0] = rx_char ;
            found_eoframe = true ;
        } else if (rx_char == '\n' && rx_control->rx_set_aside[0] == '\r') {
            rxqueue_backtrack(rx_control, 1U) ;
            rx_control->rx_set_aside[0] = rx_char ;
        } else if (rx_char == '\b' || rx_char == u'\x7f') {     // BS or DEL
            put_tx_array(uart, sizeof(bs_echo), bs_echo) ;
            rxqueue_backtrack(rx_control, 2U) ;
        } else if (rx_char == '\x15') {                         // ^U
            put_tx_char(uart, '\r') ;
            size_t blank_count = modulo_rx_queue(rx_control->rx_itor - rx_queue->head) ;
```

```
            for (size_t cnt = blank_count ; cnt != 0 ; cnt--) {
                put_tx_char(uart, ' ') ;
            }
            put_tx_char(uart, '\r') ;
            rxqueue_backtrack(rx_control, blank_count + 1) ;
        } else {
            put_tx_char(uart, rx_char) ;
            rx_control->rx_itor = modulo_rx_queue(rx_control->rx_itor + 1) ;
            rx_control->rx_set_aside[0] = rx_char ;
        }
    }

    return found_eoframe || ioqueue_full(rx_queue) ;                    // ❶
}
```

❶ A full RX queue is considered as a frame. Otherwise, there is no way to insert another character to perform line editing or terminate the input.

Line editing results in discarding characters already in the buffer when one of the special line editing characters is received. The following function performs backtracks a given number of slots in the buffer.

```
<<dev realm uart proxy: forward references>>=
static void
rxqueue_backtrack(
    UART_RxControl *const rx_control,
    size_t count) ;
```

Backtracking in the RX buffer is done by copying data “backwards” in the RX buffer. This is much like shuffling array elements to overwrite a “hole.” As long as care is taken to use modulus arithmetic when computing the indices, it is similar to a familiar byte copy. The only other complication is to recognize that the `tail` index of the RX buffer must also be moved backwards since there are fewer characters in the buffer.

Note that the most frequent case in a interactive receive of a human typing is to perform no copying at all. The line editing character will be the last item in the RX buffer and all that happens is to adjust the indices to account for discarding characters. However, if the input is being “typed” by a computer, it is quite possible to have additional characters beyond the line editing characters in the RX buffer. No matter, this function handles both cases.

```
<<dev realm uart proxy: static inline functions>>=
static void
rxqueue_backtrack(
    UART_RxControl *const rx_control,
    size_t count)
{
    assert(rx_control != NULL) ;

    UART_RxQueue *const rx_queue = &rx_control->rx_queue ;
    uint16_t *rx_storage = rx_queue->storage ;

    size_t q_count = ioqueue_count(rx_queue) ;                          // ❶
    if (q_count < count) {
        count = q_count ;
    }
    size_t src = modulo_rx_queue(rx_control->rx_itor + 1) ;             // ❷
    size_t dst = modulo_rx_queue(src - count) ;
    rx_control->rx_itor = dst ;

    while (src != rx_queue->tail) {
        rx_storage[dst] = rx_storage[src] ;
        dst = modulo_rx_queue(dst + 1) ;
        src = modulo_rx_queue(src + 1) ;
```

```
    }

    rx_queue->tail = modulo_rx_queue(rx_queue->tail - count) ;          // ❸
}
```

❶ Never backtrack more characters than there are in the RX buffer. This covers the case when a backspace is received and the RX buffer is otherwise empty.

❷ Copying starts with the character after `rx_itor`, which is positioned on the line editing character. The destination is `count` characters backwards from the `rx_itor`. Assume `count` includes the line editing character. The `rx_itor` final position is the first copied character.

❸ Having moved data backwards in the RX buffer, the `tail` index must be adjusted backwards the same amount.

With the framing transformations defined, an initialized array holds the function pointers for the various framing types.

```
<<dev realm uart proxy: static data definitions>>=
static struct {
    UART_TxFramer tx_framer ;
    UART_RxFramer rx_framer ;
} const uart_framers[uart_frame_type_count] = {
    [uart_frame_identity] = {
        .tx_framer = ident_tx_framer,
        .rx_framer = ident_rx_framer,
    },
    [uart_frame_terminal] = {
        .tx_framer = terminal_tx_framer,
        .rx_framer = terminal_rx_framer,
    },
    [uart_frame_interactive] = {
        .tx_framer = terminal_tx_framer,
        .rx_framer = interact_rx_framer,
    },
} ;
```

Apollo 3 UART Hardware Access

This section describes the code to operate on the UART peripheral device. The data sheet for the Apollo 3 UART module does not contain a sufficient description to know all the necessary rules for how the UART peripheral operates. However, from a close examination of the register descriptions it appears that the UART peripheral is an ARM PrimeCell part. The technical reference manual for the PL011 UART has been used to supplement the information in the Ambiq data sheet.

The necessary controls needed to implement data transfer across the UART involves much more than just the UART peripheral itself. Additional considerations are:

GPIO Pins
: Because a UART uses wired communications, there is a connection to the outside world. The configuration of the I/O pins used by the UART peripheral to connect to the physical transmission medium must be handled.

UART peripheral
: The UART peripheral is represented in the usual memory mapped manner.

Peripheral power
: The Apollo 3 SOC has separate power enables for peripherals which by default are turn off.

The following data structure is used to hold the specification data required to configure the manner in which the GPIO pins connected to the UART operate.

Driver_IO_Pin

```
<<dev realm uart proxy: io pin data type declaration>>=
typedef struct {
    int8_t pin_number ;
    uint8_t pad_cfg ;
} UART_IO_Pin ;
```

pin_number
The pin number. A negative pin number indicates the pin is not used.

pad_cfg
The pad configuration for the pin. This value consists of the hardware fields for the PADREG configuration of the pin. This is the only configuration information needed for UART pins.

In addition to the information about the pins which must be configured, the remainder of the information required to access the UART peripheral is gathered in the following structure.

UART_Access

```
<<dev realm uart proxy: uart access data type declaration>>=
typedef struct {
    UART0_Type *uart ;
    IRQn_Type irqn ;
    uint32_t power_enable ;
    bool hdwr_flow_support ;
    UART_IO_Pin tx ;
    UART_IO_Pin rx ;
    UART_IO_Pin rts ;
    UART_IO_Pin cts ;
} UART_Access ;
```

uart
A pointer to the base address of the UART peripheral device registers. The value is an address in memory where the UART peripheral registers are located. The data type pointed to by `uart` is the CMSIS definition of the hardware register layout for the UART.

irqn
The interrupt number for the UART interrupt. Interrupt numbers are fixed and allocated by the SOC to specific vector table offsets.

power_enable
A bit mask for the `DEVPWREN` enable register in the power controller peripheral which enables power to the UART.

hdwr_flow_support
A boolean value which indicates if the UART supports hardware flow control.

tx
The I/O pin specification for the transmit pin.

rx
The I/O pin specification for the receive pin.

rts
The I/O pin specification for the RTS pin.

cts
The I/O pin specification for the CTS pin.

When the Apollo 3 is incorporated into the Sparkfun MicroMod Artemis processor board, UART0 is attached to the USB-to-serial converter which is part of the MicroMod processor board. The only UART pins connected are those for receive and transmit lines. However, the RTS pin of the USB-to-serial converter is connected to the processor reset line. This means that when the device is opened on the host computer and the RTS line is asserted, the Apollo 3 processor is reset. This may be convenient for using a boot loader to download an executable which is the primary use case supported by the Sparkfun environment. But in our circumstances the processor reset disrupts the JLink debugger and, effectively, UART0 is not usable because of the reset.

The UART1 module is wired to the MicroMod connector as TX1, RX1, RTS1, and CTS1. This allows hardware flow control via RTS and CTS and it appears as the primary UART. The MicroMod connector also defines a RX2 and TX2 connection. These are not connected when the Artemis processor serves as in the MicroMod system processor.

This particular hardware design implies that hardware flow control may only be configured on UART1. UART0 may be configured and used for those cases where the implied reset is acceptable. The most frequently anticipated use case is for UART1 carrying output to a connected host via the USB-to-serial converter. This will provide a convenient and fast output mechanism which is independent of the SEGGER RTT mechanism.

The following table shows the pin labeling and gives the pin function select value which encodes the pin for use with the UART peripheral in a given role.

Table 9.1: UART Pin Mappings

Artemis Pin Label	Apollo 3 Pin Number	Function Select Value
TXO-0	48	0
RXI-0	49	0
TX1	12	7
RX1	25	0
RTS1	10	5
CTS1	17	7

Using the previous table, the hardware configuration information is specified as an initialized variable. Note the variable is constant as this information does not change at run time. TX and RTS pins must be configured to disable input and with no pull up resistor. RX and CTS pins must be configured to enable input, also with no pull up resistor.

The `UART_Access` structure contains configuration information which does not change at run time.

```
<<dev realm uart proxy: static data definitions>>=
static UART_Access const access_storage[UART_INSTANCES] = {
    [0] = {
        .uart = UART0,
        .irqn = UART0_IRQn,
        .power_enable = PWRCTRL_DEVPWREN_PWRUART0_Msk,
        .hdwr_flow_support = false,
        .tx = {
            .pin_number = 48,
            .pad_cfg = 0,
        },
        .rx = {
            .pin_number = 49,
            .pad_cfg = 1 << GPIO_PADREG_PAD0INPEN_Pos,
        },
        .rts = {
            .pin_number = -1,
            .pad_cfg = 0,
        },
        .cts = {
            .pin_number = -1,
            .pad_cfg = 0,
        },
    },
    [1] = {
        .uart = UART1,
        .irqn = UART1_IRQn,
        .power_enable = PWRCTRL_DEVPWREN_PWRUART1_Msk,
        .hdwr_flow_support = true,
        .tx = {
            .pin_number = 12,
            .pad_cfg = (7 << GPIO_PADREG_PAD0FNCSEL_Pos),
        },
        .rx = {
            .pin_number = 25,
            .pad_cfg = (1 << GPIO_PADREG_PAD0INPEN_Pos),
        },
        .rts = {
            .pin_number = 10,
            .pad_cfg = (5 << GPIO_PADREG_PAD0FNCSEL_Pos),
        },
        .cts = {
            .pin_number = 17,
```

```
            .pad_cfg = (7 << GPIO_PADREG_PAD0FNCSEL_Pos) |
                       (1 << GPIO_PADREG_PAD0INPEN_Pos),
        },
    },
} ;
```

The code to configure a pin for UART usage is similar to that used for GPIO's. Since the PADREG fields are 8 bits wide and packed 4 to a register, a read / insert field / write operation is necessary.

```
<<dev realm uart proxy: static function definitions>>=
static int
uart_configure_io_pin(
    UART0_Type *const uart,
    UART_IO_Pin const *const pin_config)
{
    int result = gpio_pin_alloc(pin_config->pin_number) ;
    rtcheck_zero_return(result, result) ;

    GPIO->PADKEY = GPIO_PADKEY_PADKEY_Key ;

    unsigned pin_offset ;
    uint32_t volatile *pad_reg = gpio_pad_reg(pin_config->pin_number, &pin_offset) ;
    uint32_t pad_value = *pad_reg ;
    pad_value = btwd_bits_insert(pad_value, pin_config->pad_cfg, 8, pin_offset) ;    // ❶
    *pad_reg = pad_value ;

    GPIO->PADKEY = 0 ;

    return 0 ;
}
```

❶ Yes, there are 8 bits in the pad configuration values and it is "magic."

A function to reset I/O pins to a low power default configuration is needed when the UART device is closed.

```
<<dev realm uart proxy: static function definitions>>=
static void
uart_reset_io_pin(
    UART0_Type *const uart,
    UART_IO_Pin const *const pin_config)
{
    if (pin_config->pin_number >= 0) {
        GPIO->PADKEY = GPIO_PADKEY_PADKEY_Key ;

        unsigned pin_offset ;
        uint32_t volatile *pad_reg = gpio_pad_reg(pin_config->pin_number, &pin_offset) ;
        uint32_t pad_value = *pad_reg ;

        uint32_t const gpio_fncsel = btwd_bits_insert(0, PADREG_FNCSEL_GPIO,
                PADREG_FNCSEL_WIDTH, PADREG_FNCSEL_OFFSET) ;
        pad_value = btwd_bits_insert(pad_value, gpio_fncsel, 8, pin_offset) ;
        *pad_reg = pad_value ;

        GPIO->PADKEY = 0 ;

        gpio_pin_free(pin_config->pin_number) ;
    }
}
```

Baud rate divisor computations are often quite fussy since asynchronous communications requires both sides to have reasonably precise clocks which have to be generated from the clock sources available at each end of the connection. There are a number of

schemes used by different UART peripheral designs. In the Apollo 3 case, the baud rate divisor has an integer portion of 16 bits and a fractional portion of 6 bits. In other words, the baud rate divisor is an unsigned UQ16.6 fixed binary point number. That number is stored in the hardware as two separate registers, one register for the integer portion and one for the fractional portion. The formula for computing the baud rate divisor is given in the data sheet as, $F_{UART}/(16 \times BR) = IBRD + FBRD$, where F_{UART} is the UART clock frequency and BR is the desired baud rate. The term on the right hand side of the equals sign is interpreted to mean the fixed binary point result in its separated component form. This seems to be born out by the HAL code in the SDK, albeit the calculation there is done rather strangely, indicating the author didn't have much experience in fixed radix point arithmetic. The ARM manual for the PL011 UART has a better description of the calculation on p. 3-10. The usual concern here is to avoid overflow during the calculation. The fixed binary point division is performed as a 64-bit value since multiplying the maximum UART clock frequency of 24 MHz by 64 (6 fraction bits) overflows 32 bits. Otherwise, the usual fixed binary point calculation in 64-bit variable is used and then the integer and fractional parts are separated out so they may be written to registers.

```
<<dev realm uart proxy: static function definitions>>=
static bool
uart_config_baud(
    UART0_Type *const uart,
    uint32_t baud_rate)
{
#   define  UART_BR_FRAC_BITS   6U

    static uint32_t const uart_clock_frequencies[] = {
        0, 24000000, 12000000, 6000000, 3000000, 0, 0, 0
    } ;

    uint32_t uart_clock_select = BTWD_READ_REG_FIELD(&uart->CR, UART0_CR_CLKSEL) ;  // ❶
    uint32_t uart_clock_freq = uart_clock_frequencies[uart_clock_select] ;
    if (uart_clock_freq == 0) {
        return false ;
    }

    uint64_t br_div_numer = ((uint64_t)uart_clock_freq << UART_BR_FRAC_BITS) +
            (uint64_t)btwd_field_max(UART_BR_FRAC_BITS - 1) ;                       // ❷
    uint64_t br_div_denom = (uint64_t)baud_rate * UINT64_C(16) ;
    uint32_t br_divisor = (uint32_t)(br_div_numer / br_div_denom) ;

    uint32_t i_divisor = br_divisor >> UART_BR_FRAC_BITS ;
    uint32_t f_divisor = br_divisor & btwd_field_max(UART_BR_FRAC_BITS) ;

    if (i_divisor == 0 || i_divisor > UINT16_MAX ||
            (i_divisor == UINT16_MAX && f_divisor != 0)) {                          // ❸
        return false ;
    }

    uint32_t lcrh = uart->LCRH ;                                                    // ❹
    uart->IBRD = i_divisor ;
    uart->FBRD = f_divisor ;
    uart->LCRH = lcrh ;

    return true ;

#   undef  UART_BR_FRAC_BITS
}
```

❶ Note that the UART clock rate frequency must be set before invoking this function.

❷ For the numerator of the baud rate calculation, the clock frequency is scaled by the desired number of fraction bits. The second term is a rounding term. This rounds up values greater than half the fraction amount and rounds down those half or less.

❸ Check the bounds for the integer and fractional parts. Not all combinations of baud rates and UART clock frequencies yield a valid divisor.

4. According to the ARM documentation, the LCRH, IBRD, and FBRD registers are internally a single register which is has been split out into three registers at the memory interface. The internal register is only updated when the LCRH register is written (p. 3-14). To update only the IBRD and FBRD registers, a write to LCRH must follow the updates to the IBRD and FBRD registers. Hence the pirouette of a register read followed by three register writes. The LCRH register is read to get its current value, update the IBRD and FBRD registers and follow those updates with a write to the LCRH register of its current value. Also note that as of V 2.5.1 of the Ambiq SDK, the HAL code for configuring the baud rate has a "TODO" comment about the sequence used to write the baud rate divisor registers. The code is not correct in that it does not write to LCRH, but the baud rate does get set properly since the LCRH register is written later as part of the HAL code's UART configuration.

The following are a set of small functions defined to manipulate individual controls for the UART peripheral registers. Each takes a pointer to the memory location of the UART peripheral.

```
<<dev realm uart proxy: static inline functions>>=
static inline void
enable_transmitter(
    UART0_Type *const uart)
{
    BTWD_SET_REG_FIELD(&uart->CR, UART0_CR_TXE) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline void
disable_transmitter(
    UART0_Type *const uart)
{
    BTWD_CLEAR_REG_FIELD(&uart->CR, UART0_CR_TXE) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline void
enable_tx_interrupt(
    UART0_Type *const uart)
{
    BTWD_SET_REG_FIELD(&uart->IER, UART0_IER_TXIM) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline void
disable_tx_interrupt(
    UART0_Type *const uart)
{
    BTWD_CLEAR_REG_FIELD(&uart->IER, UART0_IER_TXIM) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline bool
tx_fifo_full(
    UART0_Type *const uart)
{
    return BTWD_TEST_REG_FIELD(&uart->FR, UART0_FR_TXFF) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline bool
tx_fifo_empty(
    UART0_Type *const uart)
{
    return BTWD_TEST_REG_FIELD(&uart->FR, UART0_FR_TXFE) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline bool
tx_busy(
    UART0_Type *const uart)
{
    return BTWD_TEST_REG_FIELD(&uart->FR, UART0_FR_BUSY) ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline void
put_tx_char(
    UART0_Type *const uart,
    uint8_t tx_char)
{
    while (tx_fifo_full(uart)) {
        // empty ;
    }
    uart->DR = tx_char ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline void
put_tx_array(
    UART0_Type *const uart,
    size_t size,
    uint8_t tx_string[size])
{
    for ( ; size != 0 ; size--) {
        put_tx_char(uart, *tx_string++) ;
    }
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline void
enable_rx_interrupt(
    UART0_Type *const uart)
{
    uint32_t ier = uart->IER ;
    ier = BTWD_FIELD_SET(ier, UART0_IER_RXIM) ;
    ier = BTWD_FIELD_SET(ier, UART0_IER_RTIM) ;
    uart->IER = ier ;
}
```

```
<<dev realm uart proxy: static inline functions>>=
static inline bool
rx_fifo_empty(
    UART0_Type *const uart)
{
    return BTWD_TEST_REG_FIELD(&uart->FR, UART0_FR_RXFE) ;
}
```

Apollo 3 UART Instantiation

The Apollo 3 has two UART modules.

```
<<dev realm uart param: constants>>=
#define UART_INSTANCES  2
```

The UART device control blocks are stored as an initialized array.

```
<<dev realm uart proxy: uart instances definitions>>=
static UART_ControlBlock
uart_control_blocks[UART_INSTANCES] = {
    [0] = {
        .tx_control = {
            .tx_queue = {
                .head = 0,
                .tail = 0,
                .storage = {0},
            },
            .tx_framer = NULL,
            .tx_set_aside = {0},
            .tx_itor = 0,
            .tx_short_write = false,
        },
        .rx_control = {
            .rx_queue = {
                .head = 0,
                .tail = 0,
                .storage = {0},
            },
            .rx_framer = NULL,
            .rx_set_aside = {0},
            .rx_itor = 0,
            .rx_short_read = false,
        },
        .access = &access_storage[0],
        .proxy = NULL,
    },
    [1] = {
        .tx_control = {
            .tx_queue = {
                .head = 0,
                .tail = 0,
                .storage = {0},
            },
            .tx_framer = NULL,
            .tx_set_aside = {0},
            .tx_itor = 0,
            .tx_short_write = false,
        },
        .rx_control = {
            .rx_queue = {
                .head = 0,
                .tail = 0,
                .storage = {0},
            },
            .rx_framer = NULL,
            .rx_set_aside = {0},
            .rx_itor = 0,
            .rx_short_read = false,
        },
        .access = &access_storage[1],
        .proxy = NULL,
    },
} ;
```

Servicing the UART

This section discusses the foreground processing required to service the UART. Because there is no DMA associated with the UART on the Apollo 3, the processor executes code to move data to and from the UART FIFO's. The necessary code starts with the IRQ handler.

UART IRQ Handling

Each UART peripheral is tied to a single IRQ in the processor core. So, the IRQ handler must deal with both receiving and transmitting. All the common code has been factored into a single function, so the IRQ handers themselves are simple one-liners.

```
<<dev realm uart proxy: external function definitions>>=
void
UART0_IRQHandler(void)
{
    uart_irq_handler(0) ;
}
```

```
<<dev realm uart proxy: external function definitions>>=
void
UART1_IRQHandler(void)
{
    uart_irq_handler(1) ;
}
```

This design uses only three interrupt sources in the UART. The receive and transmit FIFO interrupts indicate when the FIFO needs service. The receive timeout interrupt indicates that there are bytes in the receive FIFO and there has been a time gap where no additional byte has arrived. Note neither interrupts for error conditions are not enabled nor for modem control lines status changes. Since each received character carries with it 4 bits of error information and since the error information is tracked through the RX queue, errors are processed as each byte is read out of the RX queue. Since only simplified RTS/CTS hardware flow control is supported (and not complete modem control which might require RI, DCD and DSR status), the peripheral itself handles the necessary changes in the I/O pins associated with the flow control.

```
<<dev realm uart proxy: static function definitions>>=
static void
uart_irq_handler(
    SVC_DevInstance inst)
{
    assert(inst < UART_INSTANCES) ;

    UART_ControlBlock *const ucb = &uart_control_blocks[inst] ;
    UART0_Type *const uart = ucb->access->uart ;

    uint32_t const rx_irq_mask = UART0_MIS_RTMIS_Msk | UART0_MIS_RXMIS_Msk ;
    uint32_t irq_sources = uart->MIS ;
    if ((irq_sources & rx_irq_mask) != 0) {
        ReceiveRxData(ucb) ;
    }

    if ((irq_sources & UART0_MIS_TXMIS_Msk) != 0) {
        TransmitTxData(ucb) ;
    }

    uart->IEC = irq_sources ;
}
```

The processing of interrupts is simplified somewhat by the fact that the peripheral has a masked interrupt status which indicates directly which of the enabled sources caused the interrupt. Note that the RX FIFO interrupt and RX timeout interrupt sources execute the same code, *i.e.* receive bytes from the RX FIFO into the RX queue.

UART Transmit Operations

This section shows the operations used for transmitting. There is one function used by the IRQ handler for the transmit side. This function is the core of what must happen to transfer bytes from the transmit buffering scheme to the UART transmit FIFO. The transmit FIFO is filled to capacity as long as there is sufficient data in the buffer.

```
<<dev realm uart proxy: forward references>>=
static void TransmitTxData(
    UART_ControlBlock *const ucb) ;
```

ucb
A pointer to a UART control block corresponding to the UART on which characters are to be transmitted.

The `TransmitTxData` function reads character data from the RX queue and writes it in the TX FIFO. Data is transmitted until either the FIFO is full or there are no remaining characters in the TX queue.

```
<<dev realm uart proxy: static function definitions>>=
static void
TransmitTxData(
    UART_ControlBlock *const ucb)
{
    assert(ucb != NULL) ;

    size_t tx_count = fill_tx_fifo(ucb) ;

    if (ucb->tx_control.tx_short_write && tx_count != 0 &&
            notify_to_background(ucb, uart_notify_tx_available)) {
        ucb->tx_control.tx_short_write = false ;
    }
}
```

There are only two steps:

- Fill the TX FIFO from the TX queue, recording the number of bytes placed in the FIFO.
- Queue a background notification based on:
  - Past history as to whether the last transmit request was fulfilled.
  - The number of bytes just placed in the TX FIFO.

Note that the logic allows for the background notification to fail, *i.e.* there was no room in the background notification queue. In that case, the `tx_short_write` status is maintained as `true` so another attempt at the notification is made.

The semantics of the notifications are discussed below.

```
<<dev realm uart proxy: forward references>>=
static size_t
fill_tx_fifo(
    UART_ControlBlock *const ucb) ;
```

Filling the TX FIFO happens as expected. Characters are transferred from the TX queue to the FIFO until either there are no more characters in the TX queue or the FIFO is full.

```
<<dev realm uart proxy: static function definitions>>=
static size_t
fill_tx_fifo(
    UART_ControlBlock *const ucb)
{
```

```
    UART0_Type *const uart = ucb->access->uart ;
    UART_TxQueue *tx_queue = &ucb->tx_control.tx_queue ;

    uint8_t tx_char ;
    size_t tx_count = 0 ;
    for ( ; !tx_fifo_full(uart) && ioqueue_remove(tx_queue, &tx_char) ;
            tx_count++) {
        uart->DR = tx_char ;
    }

    return tx_count ;
}
```

UART Receive Operations

One wishes that receiving was just like transmitting only with some conceptual flow switched in the opposite direction. Unfortunately, that is almost never the case. For UART communications, receiving is more complicated because:

- The receiver can detect errors in the received data, *e.g.* framing errors. The transmit side simply dumps bits onto a wire and hopes for the best.
- Receiving means data is being *pushed* at you and it is necessary to find the space to put it. Hardware flow of control can help this issue, but in its absence, the only option is to supply some buffering for incoming data and be prepared for overflow if the transmitting peer is insistent.
- The receiver has an internal FIFO which holds 32 bytes of input before it overflows.
- The UART has a built-in timeout that is triggered when the receive FIFO contains data and 32-bit times have elapsed. This is useful for handling received characters where there have not been a sufficient number to trigger the FIFO threshold interrupt. As shown in the IRQ handler, the receive timeout is handled in the same manner as the receive FIFO interrupt, *i.e.* move bytes from the receive FIFO and into the RX queue.

Despite these differences from the transmit side, reception is conceived of as requesting characters from the RX queue which is filled by the IRQ handler from the UART FIFO.

Receive UART Operations

```
<<dev realm uart proxy: forward references>>=
static void ReceiveRxData(
    UART_ControlBlock *const ucb) ;
```

ucb
A pointer to a UART control block corresponding to the UART where characters are to be received.

The `ReceiveRxData` function reads character data from the UART FIFO and places it in the RX queue. Data is received until either the FIFO is empty or there is no remaining space in the RX queue. If in the process of receiving data a serial line *break* condition is detected, then a notification for the break is placed in the Background Notification Queue. If a characters is received in error, the received data is placed in the RX queue, and the error is handled when the data is read out of the RX queue.

```
<<dev realm uart proxy: static function definitions>>=
static void
ReceiveRxData(
    UART_ControlBlock *const ucb)
{
```

```
    assert(ucb != NULL) ;

    UART_RxQueue *rx_queue = &ucb->rx_control.rx_queue ;
    UART0_Type *const uart = ucb->access->uart ;

    unsigned rx_count = 0 ;
    for ( ; !(rx_fifo_empty(uart) || ioqueue_full(rx_queue)) ; rx_count++) {
        uint16_t rx_char = (uint16_t)uart->DR ;

        if (btwd_field_test(rx_char, UART0_DR_BEDATA_Msk)) {
            (void)notify_to_background(ucb, uart_notify_break) ;
        } else {
            bool inserted = ioqueue_insert(rx_queue, rx_char) ;
            assert(inserted) ; (void)inserted ;
        }
    }

    if (ucb->rx_control.rx_short_read && rx_count != 0 &&
            notify_to_background(ucb, uart_notify_rx_available)) {
        ucb->rx_control.rx_short_read = false ;
    }
}
```

UART Notifications

The UART device issues three types of notifications.

```
<<dev realm uart param: data type declarations>>=
typedef enum {
    uart_notify_tx_available,
    uart_notify_rx_available,
    uart_notify_break,
} UART_NotifyType ;
```

uart_notify_tx_available
: The `uart_notify_tx_available` notification type is issued to the background when the last transmit request filled the TX queue and there is now more space available in the queue. Transmit requests which are *not* completely transferred to the TX queue receive a TX available notification to know when to transmit additional data.

uart_notify_rx_available
: The `uart_notify_rx_available` notification type is issued to the background when the last receive request was *not* fulfilled because the RX queue was emptied. The notification is an indication to background processing that additional bytes have been received.

uart_notify_break
: The `uart_notify_break` notification type is issued when the UART peripheral detects a **break** condition on the serial line. This notification is queued to the background as *out of band* data, *i.e.* the notification is sent when the break is read off of the UART peripheral and is not sent through the RX queue.

As mentioned previously, each received character has an associated error status which is tracked along with the character in the RX queue. Error status is returned when the byte received in error is transferred out of the RX queue. The value of the received byte is passed to the background. Reading a error byte ends the request and passes back the error status. The error status is represented as a bit encoded enumeration.

```
<<dev realm uart param: data type declarations>>=
typedef enum {
    uart_success = 0,
```

```
    uart_frame_err = 1,
    uart_parity_err = 2,
    uart_overrun_err = 8,
} UART_RxErrStatus ;                    // ❶
```

❶ Note this enumeration is contrived to have the same bit value encoding as obtained from the UART receive status register. Also, the break status has been omitted in this enumeration because the break condition is notified asynchronously and out of band.

The notification data structure for the UART includes the number of bytes in both the RX and TX queues.

SVC_DevUartNotification

```
<<dev realm uart param: data type declarations>>=
DECLARE_DEV_NOTIFICATION(SVC_DevUartNotification,
    UART_NotifyType type ;
    size_t tx_count ;
    size_t rx_count ;
) ;
```

The background notification proxy takes an argument of the device specific form of the notification.

```
<<dev realm uart param: data type declarations>>=
typedef void (*SVC_DevUartNotifyProxy)(
        SVC_DevUartNotification const *const params) ;
```

All the above considerations are factored into a function that queues a UART buffer notification into the Background Notification Queue.

```
<<dev realm uart proxy: forward references>>=
static bool
notify_to_background(
    UART_ControlBlock *const ucb,
    UART_NotifyType type) ;
```

ucb
A pointer to a UART control block.

type
The type of UART notification to queue.

The `notify_to_background` function queues a UART notification in the background notification queue. The return value is `true` if the notification was successfully queued and `false` otherwise.

N.B. failure to queue a UART notification to the Background Notification Queue is *not* considered a "panic" condition. This recognizes the *interactive* nature of computer-to-computer communications. Missing a notification gives the two communicating peers a changes to catch up with each other. Furthermore, an overly exuberant transmitter must *not* to be able to force a panic condition on the system.

```
<<dev realm uart proxy: static function definitions>>=
static bool
notify_to_background(
    UART_ControlBlock *const ucb,
    UART_NotifyType type)
{
    assert(ucb != NULL) ;
```

```
    SVC_DevUartNotifyProxy proxy = ucb->proxy ;
    assert(proxy != NULL) ;

    SVC_DevUartNotification *notification =
            probe_bg_queue(sizeof(SVC_DevUartNotification)) ;
    if (notification == NULL) {
        return false ;
    }

    notification->block_size = sizeof(SVC_DevUartNotification) ;
    notification->notify_proxy = (SVC_DevNotifyProxy)proxy ;
    notification->notify_closure = ucb->closure ;
    notification->device_class = DEV_UART_CLASS ;
    notification->device_instance = ucb - uart_control_blocks ;
    notification->type = type ;
    notification->tx_count = ioqueue_count(&ucb->tx_control.tx_queue) ;
    notification->rx_count = ioqueue_count(&ucb->rx_control.rx_queue) ;

    bool pushed = push_bg_queue() ;
    if (!pushed) {
        panic("failed to push UART notification") ;                // ❶
    }

    return pushed ;
}
```

❶ Once “probed,” failure to “push” the background notification queue is a panic condition.

UART Background Requests

The interface to the UART device for background processing is provided by five functions:

- Open the UART
- Close the UART
- Write data to transmit
- Read received data
- Query status

```
<<dev realm uart param: data type declarations>>=
enum SVC_uartRequest {
    uart_open = 0,
    uart_close,
    uart_transmit,
    uart_receive,
    uart_query,

    uart_operation_count        // last
} ;
```

The following diagram shows the combination of peripheral devices used to implement the logical UART device. Notice that in addition to the expected interactions with the NVIC and the UART peripheral, the UART device also uses the GPIO, Power Controller, and System Timer peripherals. The GPIO peripheral is used to set up the pin which connect the UART to the

environment. The Power Controller controls the power domain associated with the UART peripheral. There are more than the usual interactions for the UART device.

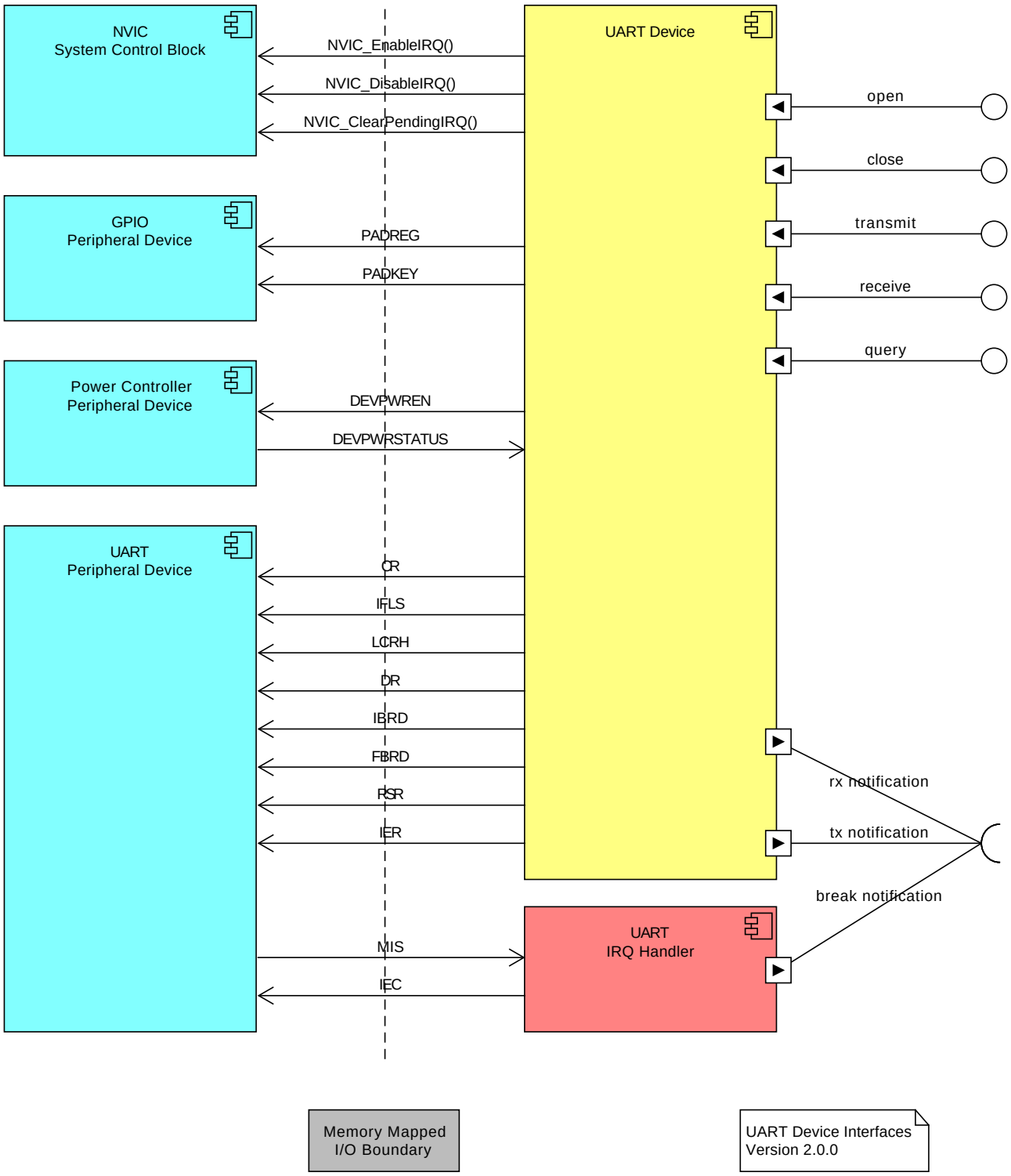

Figure 9.2: Snapshot of UART Interface Components

Open a UART

```
<<dev svc uart req: external declarations>>=
extern int
dev_uart_open(
    SVC_DevInstance uart,
    uint32_t baud_rate,
    bool hdwr_flow_ctrl,
    UART_FramingType framing,
    SVC_DevUartNotifyProxy proxy,
    SVC_DevNotifyClosure closure) ;
```

uart
: The instance number which identifies which UART device is to be opened.

baud_rate
: The desired baud rate of the UART.

hdwr_flow_ctrl
: A boolean to indicate if hardware flow control is to be used.

proxy
: A pointer to a notification proxy function which is to be invoked during background processing when a UART buffer changes status.

closure
: A closure data value which is included in any background notifications.

The `dev_uart_open` function makes ready the UART given by, `uart`. This function must be invoked before any other operations on the UART.

An appropriate data structure is needed to transfer input parameters across the `SVC` interface.

```
<<dev realm uart param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevUartOpenInput,
    uint32_t baud_rate ;
    bool hdwr_flow_ctrl ;
    UART_FramingType framing ;
    SVC_DevUartNotifyProxy proxy ;
    SVC_DevNotifyClosure closure ;
) ;
```

The background functions which marshal the request follow the same pattern used previously, fill in the parameter structures and make a device realm `SVC` call.

```
<<dev svc uart req: external definitions>>=
int
dev_uart_open(
    SVC_DevInstance uart,
    uint32_t baud_rate,
    bool hdwr_flow_ctrl,
    UART_FramingType framing,
    SVC_DevUartNotifyProxy proxy,
    SVC_DevNotifyClosure closure)
{
    SVC_DevUartOpenInput input = {
        .block_size = sizeof(SVC_DevUartOpenInput),
        .baud_rate = baud_rate,
        .hdwr_flow_ctrl = hdwr_flow_ctrl,
```

```
        .framing = framing,
        .proxy = proxy,
        .closure = closure,
    } ;

    SVC_DevRequest uart_req = dev_req_encode(DEV_UART_CLASS, uart_open, uart) ;
    return dev_realm_svc_call(uart_req, &input, NULL, NULL) ;
}
```

The foreground proxy function for opening the UART is long because there are multiple peripherals involved. It must handle:

- Allocation and configuration of the I/O pins used by the UART.
- Enabling power to the UART from the power controller.
- Configuring the UART itself.
- Initializing the control block data used to manage the UART operations.
- Enabling the UART and its interrupt.

The following function is the foreground proxy for `dev_uart_open`.

```
<<dev realm uart proxy: static function definitions>>=
static int
dev_realm_uart_open(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    rtcheck_max_return(inst, UART_INSTANCES, -ERR_INVALID_PARAM) ;
    UART_ControlBlock *const ucb = &uart_control_blocks[inst] ;
    rtcheck_NULL_return(ucb->proxy, -ERR_OPERATION_FAILED) ;        // ❶

    UART_Access const *const uart_access = ucb->access ;
    UART0_Type *const uart = uart_access->uart ;

    <<dev_realm_uart_open: get arguments>>
    <<dev_realm_uart_open: configure I/O pins>>
    <<dev_realm_uart_open: enable power>>
    <<dev_realm_uart_open: configure uart>>
    <<dev_realm_uart_open: init control block>>
    <<dev_realm_uart_open: enable uart>>

    return 0 ;

    <<dev_realm_uart_open: error exit>>
}
```

❶ The device is already open if it has a non-NULL proxy function pointer.

The input arguments must be read in from unprivileged memory. Some validation is necessary since a request for hardware flow control may be made only for a UART that supports it and a proxy function must be specified to handle the buffer notifications.

```
<<dev_realm_uart_open: get arguments>>=
SVC_DevUartOpenInput open_input ;
int result = copy_in_svc_param(input, sizeof(open_input), &open_input) ;
rtcheck_zero_return(result, result) ;

if ((open_input.hdwr_flow_ctrl && !uart_access->hdwr_flow_support) ||
        (open_input.proxy == NULL)) {
    return -ERR_INVALID_PARAM ;
}

rtcheck_return(open_input.framing < uart_frame_type_count, -ERR_INVALID_PARAM) ;
```

```
<<dev_realm_uart_open: configure I/O pins>>=
bool configured = 0 == uart_configure_io_pin(uart, &uart_access->rx) ;
configured = configured && (0 == uart_configure_io_pin(uart, &uart_access->tx)) ;
if (open_input.hdwr_flow_ctrl) {
    configured = configured && (0 == uart_configure_io_pin(uart, &uart_access->rts)) ;
    configured = configured && (0 == uart_configure_io_pin(uart, &uart_access->cts)) ;
}
if (!configured) {
    result = -ERR_OPERATION_FAILED ;
    goto error_exit ;                                   // ❶
}
```

❶ Before all the `goto` naysayers have apoplexy, note this usage is forward jumping only for resource clean up. If there were exception or deferral mechanisms in "C", then those mechanisms would be used. Yes, there is always a way to code it without a `goto`, but not without considerable other clutter.

The Apollo 3 SOC has power controls for all the peripherals. To use the UART, it must be powered up and execution flow must wait for the power up status to show. The wait time is short, so a busy loop is used. The loop counting is to insure loop termination if the UART power status never shows on.

```
<<dev_realm_uart_open: enable power>>=
PWRCTRL->DEVPWREN |= uart_access->power_enable ;
unsigned cnt = 1000000 ;                            // just some large number
while (cnt-- != 0) {
    if ((PWRCTRL->DEVPWRSTATUS & PWRCTRL_DEVPWRSTATUS_HCPA_Msk) != 0) {
        break ;
    }
}

if (cnt == 0) {
    PWRCTRL->DEVPWREN &= ~uart_access->power_enable ;
    result = -ERR_OPERATION_FAILED ;
    goto error_exit ;
}
```

Much of the configuration of the UART is fixed by the manner in which the peripheral is operated. The FIFO's are set to trigger at 3/4 full for receiving and 1/4 remaining for transmit.

```
<<dev_realm_uart_open: configure uart>>=
uart->CR = 0 ;
uart->IER = 0 ;
uart->IEC = btwd_field_max(UART0_IEC_OEIC_Pos) ;

uint32_t ifls_reg = 0 ;
ifls_reg = BTWD_FIELD_INSERT(ifls_reg, UART0_IFLS_RXIFLSEL_FIFO_3_4,
        UART0_IFLS_RXIFLSEL) ;
ifls_reg = BTWD_FIELD_INSERT(ifls_reg, UART0_IFLS_TXIFLSEL_FIFO_1_4,
```

```
        UART0_IFLS_TXIFLSEL) ;
uart->IFLS = ifls_reg ;

uint32_t lcrh_reg = 0 ;
lcrh_reg = BTWD_FIELD_INSERT(lcrh_reg, UART0_LCRH_WLEN_8_BIT_CHAR,
        UART0_LCRH_WLEN) ;
lcrh_reg = BTWD_FIELD_SET(lcrh_reg, UART0_LCRH_FEN) ;
uart->LCRH = lcrh_reg ;

uint32_t cr_reg = uart->CR ;
cr_reg = BTWD_FIELD_SET(cr_reg, UART0_CR_CLKEN) ;
cr_reg = BTWD_FIELD_INSERT(cr_reg, UART0_CR_CLKSEL_24MHZ, UART0_CR_CLKSEL) ;
if (open_input.hdwr_flow_ctrl) {
    cr_reg = BTWD_FIELD_SET(cr_reg, UART0_CR_CTSEN) ;
    cr_reg = BTWD_FIELD_SET(cr_reg, UART0_CR_RTSEN) ;
}
cr_reg = BTWD_FIELD_SET(cr_reg, UART0_CR_RXE) ;
cr_reg = BTWD_FIELD_SET(cr_reg, UART0_CR_TXE) ;
uart->CR = cr_reg ;

bool baud_configured = uart_config_baud(uart, open_input.baud_rate) ;
if (!baud_configured) {
    result = -ERR_OPERATION_FAILED ;
    goto error_exit ;
}
```

The I/O queues for both receiving and transmitting are cleared out. The remainder of the control block members are also initialized.

```
<<dev_realm_uart_open: init control block>>=
ioqueue_init(&ucb->tx_control.tx_queue) ;
ucb->tx_control.tx_framer = uart_framers[open_input.framing].tx_framer ;
memset(ucb->tx_control.tx_set_aside, 0, sizeof(ucb->tx_control.tx_set_aside)) ;
ucb->tx_control.tx_itor = 0 ;
ucb->tx_control.tx_short_write = false ;

ioqueue_init(&ucb->rx_control.rx_queue) ;
ucb->rx_control.rx_framer = uart_framers[open_input.framing].rx_framer ;
memset(ucb->rx_control.rx_set_aside, 0, sizeof(ucb->rx_control.rx_set_aside)) ;
ucb->rx_control.rx_itor = 0 ;
ucb->rx_control.rx_short_read = false ;

ucb->proxy = open_input.proxy ;
ucb->closure = open_input.closure ;
```

Finally, the UART is enabled and its interrupts primed.

```
<<dev_realm_uart_open: enable uart>>=
cr_reg = BTWD_FIELD_SET(cr_reg, UART0_CR_UARTEN) ;
uart->CR = cr_reg ;

enable_tx_interrupt(uart) ;
enable_rx_interrupt(uart) ;
NVIC_ClearPendingIRQ(uart_access->irqn) ;
NVIC_EnableIRQ(uart_access->irqn) ;
```

If errors occur, all the I/O pins must be reset. They are configured for GPIO functionality and deallocated.

```
<<dev_realm_uart_open: error exit>>=
error_exit:
    uart_reset_io_pin(uart, &uart_access->cts) ;
    uart_reset_io_pin(uart, &uart_access->rts) ;
```

```
    uart_reset_io_pin(uart, &uart_access->rx) ;
    uart_reset_io_pin(uart, &uart_access->tx) ;

    return result ;
```

Close a UART

```
<<dev svc uart req: external declarations>>=
extern int
dev_uart_close(
    SVC_DevInstance uart) ;
```

uart
The instance number which identifies which UART device is to be closed.

The `dev_uart_close` function shuts down the UART given by, `uart`.

The background request function for closing a UART is trivial.

```
<<dev svc uart req: external definitions>>=
int
dev_uart_close(
    SVC_DevInstance uart)
{
    SVC_DevRequest uart_req = dev_req_encode(DEV_UART_CLASS, uart_close, uart) ;
    return dev_realm_svc_call(uart_req, NULL, NULL, NULL) ;
}
```

Closing a UART is simpler than opening one. The following function is the foreground proxy for `dev_uart_close`.

```
<<dev realm uart proxy: static function definitions>>=
static int
dev_realm_uart_close(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(input == NULL) ; (void)input ;
    assert(output == NULL) ; (void)output ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    rtcheck_max_return(inst, UART_INSTANCES, -ERR_INVALID_PARAM) ;
    UART_ControlBlock *const ucb = &uart_control_blocks[inst] ;
    rtcheck_not_NULL_return(ucb->proxy, -ERR_OPERATION_FAILED) ;

    UART_Access const *const uart_access = ucb->access ;

    NVIC_DisableIRQ(uart_access->irqn) ;

    UART0_Type *const uart = uart_access->uart ;
    uart->CR = 0 ;
    uart->IER = 0 ;
    uart->IEC = btwd_field_max(UART0_IEC_OEIC_Pos) ;

    PWRCTRL->DEVPWREN &= ~uart_access->power_enable ;
```

```
    uart_reset_io_pin(uart, &uart_access->cts) ;
    uart_reset_io_pin(uart, &uart_access->rts) ;
    uart_reset_io_pin(uart, &uart_access->rx) ;
    uart_reset_io_pin(uart, &uart_access->tx) ;

    ucb->proxy = NULL ;

    return 0 ;
}
```

UART Transmit

```
<<dev svc uart req: external declarations>>=
extern int
dev_uart_transmit(
    SVC_DevInstance uart,
    size_t capacity,
    uint8_t const buffer[capacity]) ;
```

uart
: The instance number which identifies which UART device is to transmit the contents of `buffer`.

capacity
: The number of bytes pointed to by `buffer`.

buffer
: A pointer to the characters to be transmitted.

The `dev_uart_transmit` function requests `capacity` number of bytes pointed to by `buffer` be transmitted on the UART given by, `uart`. The return value of the function is negative if an error occurred. Non-negative return values indicate the number of bytes from `buffer` actually queued for transmission. If the return number of queue bytes is less than `capacity`, then the caller must retry to transmit the remaining bytes. The caller can know when to retry *short* writes by waiting for the `uart_tx_available` type of UART notification.

An appropriate data structure is required to transfer input parameters across the `SVC` interface.

```
<<dev realm uart param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevUartTransmitInput,
    size_t capacity ;
    uint8_t const *buffer ;
) ;
```

```
<<dev realm uart param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevUartTransmitOutput,
    size_t count ;
) ;
```

The background functions which marshal the request follow the same pattern used previously, fill in the parameter structures and make a device realm `SVC` call.

```
<<dev svc uart req: external definitions>>=
int
dev_uart_transmit(
    SVC_DevInstance uart,
    size_t capacity,
    uint8_t const buffer[capacity])
```

```
{
    SVC_DevUartTransmitInput input = {
        .block_size = sizeof(SVC_DevUartTransmitInput),
        .capacity = capacity,
        .buffer = buffer,
    } ;

    SVC_DevUartTransmitOutput output = {
        .block_size = sizeof(SVC_DevUartTransmitOutput),
        .count = 0,
    } ;

    SVC_DevRequest uart_req = dev_req_encode(DEV_UART_CLASS, uart_transmit, uart) ;
    int status = dev_realm_svc_call(uart_req, &input, &output, NULL) ;

    return status < 0 ? status : (int)output.count ;
}
```

The following function is the foreground proxy for `dev_uart_transmit`.

```
<<dev realm uart proxy: static function definitions>>=
static int
dev_realm_uart_transmit(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    rtcheck_max_return(inst, UART_INSTANCES, -ERR_INVALID_PARAM) ;
    UART_ControlBlock *const ucb = &uart_control_blocks[inst] ;

    SVC_DevUartTransmitInput trans_input ;
    int result = copy_in_svc_param(input, sizeof(trans_input), &trans_input) ;
    rtcheck_zero_return(result, result) ;
    rtcheck_not_NULL_return(trans_input.buffer, -ERR_INVALID_PARAM) ;
    rtcheck_not_zero_return(trans_input.capacity, -ERR_INVALID_PARAM) ;

    SVC_DevUartTransmitOutput trans_output = {
        .block_size = sizeof(trans_output),
        .count = 0,
    } ;

    IRQn_Type irq_num = ucb->access->irqn ;
    (void)stwd_begin_interrupt_section(irq_num) ;
//BEGIN INTERRUPT SECTION

    trans_output.count = ucb->tx_control.tx_framer(ucb,
            trans_input.capacity, trans_input.buffer) ;
    (void)fill_tx_fifo(ucb) ;

    stwd_end_interrupt_section(irq_num, true) ;
//END INTERRUPT SECTION

    if (trans_output.count < trans_input.capacity) {
        ucb->tx_control.tx_short_write = true ;
    }

    return copy_out_svc_result(output, sizeof(trans_output), &trans_output) ;
}
```

UART Receive

```
<<dev svc uart req: external declarations>>=
extern int
dev_uart_receive(
    SVC_DevInstance uart,
    size_t capacity,
    uint8_t buffer[capacity],
    UART_RxErrStatus *const status_ref) ;
```

uart
The instance number which identifies which UART device is to receive data into `buffer`.

capacity
The number of bytes pointed to by `buffer`.

buffer
A pointer to the memory where received characters are placed.

status_ref
A pointer to a memory object where the receive error status is placed.

The `dev_uart_receive` function attempts to read `capacity` number of bytes from the receive queue of `uart` and places any bytes read into the memory pointed to by `buffer`.

If the return value is negative, then an error occurred. Non-negative return values give the number of bytes placed in `buffer`. If the returned number of received bytes is less than `capacity`, then the caller must invoke `dev_uart_receive` again to obtain any additional data. The caller can know when to retry *short* reads by waiting for the `uart_rx_available` type of UART notification.

After returning from the function, the memory pointed to by `status_ref` contains the error status of the read. If the status is `uart_success`, then all the transferred bytes were received without error. If the status is otherwise, then it indicates the cause of the receive error and the last byte placed in buffer is the one in error. The value of the byte in error is returned, *i.e.* `buffer[count - 1]`, where `count` is a positive return value of the function, is the byte as it was received by the UART peripheral. Receive requests are ended, possibly short, when the first byte encountered is found to be in error.

```
<<dev realm uart param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevUartReceiveInput,
    size_t capacity ;
    uint8_t *buffer ;
) ;
```

```
<<dev realm uart param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevUartReceiveOutput,
    size_t count ;
    UART_RxErrStatus status ;
) ;
```

```
<<dev svc uart req: external definitions>>=
int
dev_uart_receive(
    SVC_DevInstance uart,
    size_t capacity,
    uint8_t buffer[capacity],
    UART_RxErrStatus *const status_ref)
{
    SVC_DevUartReceiveInput input = {
        .block_size = sizeof(SVC_DevUartReceiveInput),
```

```
        .capacity = capacity,
        .buffer = buffer,
    } ;

    SVC_DevUartReceiveOutput output = {
        .block_size = sizeof(SVC_DevUartReceiveOutput),
        .count = 0,
        .status = uart_success,
    } ;

    SVC_DevRequest uart_req = dev_req_encode(DEV_UART_CLASS, uart_receive, uart) ;
    int status = dev_realm_svc_call(uart_req, &input, &output, NULL) ;
    rtcheck_zero_return(status, status) ;

    if (status_ref != NULL) {
        *status_ref = output.status ;
    }
    return (int)output.count ;
}
```

The following function is the foreground proxy for `dev_uart_receive`.

```
<<dev realm uart proxy: static function definitions>>=
static int
dev_realm_uart_receive(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    rtcheck_max_return(inst, UART_INSTANCES, -ERR_INVALID_PARAM) ;
    UART_ControlBlock *const ucb = &uart_control_blocks[inst] ;
    rtcheck_not_NULL_return(ucb->proxy, -ERR_OPERATION_FAILED) ;

    SVC_DevUartReceiveInput recv_input ;
    int result = copy_in_svc_param(input, sizeof(recv_input), &recv_input) ;
    rtcheck_zero_return(result, result) ;
    rtcheck_not_NULL_return(recv_input.buffer, -ERR_INVALID_PARAM) ;
    rtcheck_not_zero_return(recv_input.capacity, -ERR_INVALID_PARAM) ;

    SVC_DevUartReceiveOutput recv_output = {
        .block_size = sizeof(recv_output),
        .count = 0,
        .status = uart_success,
    } ;

    IRQn_Type irq_num = ucb->access->irqn ;
    (void)stwd_begin_interrupt_section(irq_num) ;
//BEGIN INTERRUPT SECTION

    recv_output.count = ucb->rx_control.rx_framer(ucb, recv_input.capacity,
            recv_input.buffer, &recv_output.status) ;

    stwd_end_interrupt_section(irq_num, true) ;
//END INTERRUPT SECTION

    if (recv_output.count < recv_input.capacity) {
        ucb->rx_control.rx_short_read = true ;
    }
```

```
    return copy_out_svc_result(output, sizeof(recv_output), &recv_output) ;
}
```

UART Query

```
<<dev svc uart req: external declarations>>=
extern int
dev_uart_query(
    SVC_DevInstance uart,
    size_t *rx_count_ref,
    size_t *tx_count_ref) ;
```

uart
: The instance number which identifies which UART device is to be flushed.

rx_count_ref
: A pointer to a memory object to which the number of bytes in the RX queue is written. If the value of `rx_count_ref` is `NULL`, then no receive queue count is returned.

tx_count_ref
: A pointer to a memory object to which the number of bytes in the TX queue is written. If the value of `xx_count_ref` is `NULL`, then no transmit queue count is returned.

The `dev_uart_queue` function queries the number of bytes in the RX and TX queues for the UART given by, `uart`. The return value is 0 for success and negative otherwise. The returned counts are returned by reference via the `rx_count_ref` and `tx_count_ref` pointers.

```
<<dev realm uart param: data type declarations>>=
DECLARE_REQUEST_PARAM(SVC_DevUartQueryOutput,
    size_t rx_count ;
    size_t tx_count ;
) ;
```

```
<<dev svc uart req: external definitions>>=
int
dev_uart_query(
    SVC_DevInstance uart,
    size_t *rx_count_ref,
    size_t *tx_count_ref)
{
    SVC_DevUartQueryOutput output = {
        .block_size = sizeof(SVC_DevUartQueryOutput),
        .rx_count = 0,
        .tx_count = 0,
    } ;

    SVC_DevRequest uart_req = dev_req_encode(DEV_UART_CLASS, uart_query, uart) ;
    int status = dev_realm_svc_call(uart_req, NULL, &output, NULL) ;
    rtcheck_zero_return(status, status) ;

    if (rx_count_ref != NULL) {
        *rx_count_ref = output.rx_count ;
    }
    if (tx_count_ref != NULL) {
        *tx_count_ref = output.tx_count ;
    }
```

```
    return 0 ;
}
```

```
<<dev realm uart proxy: static function definitions>>=
static int
dev_realm_uart_query(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    assert(input == NULL) ; (void)input ;
    assert(error == NULL) ; (void)error ;

    SVC_DevInstance inst = dev_req_extract_instance(req) ;
    rtcheck_max_return(inst, UART_INSTANCES, -ERR_INVALID_PARAM) ;
    UART_ControlBlock *const ucb = &uart_control_blocks[inst] ;
    rtcheck_not_NULL_return(ucb->proxy, -ERR_OPERATION_FAILED) ;

    SVC_DevUartQueryOutput query_output ;
    query_output.block_size = sizeof(SVC_DevUartQueryOutput) ;

    IRQn_Type irq_num = ucb->access->irqn ;
    (void)stwd_begin_interrupt_section(irq_num) ;
//BEGIN INTERRUPT SECTION

    query_output.rx_count = ioqueue_count(&ucb->rx_control.rx_queue) ;
    query_output.tx_count = ioqueue_count(&ucb->tx_control.tx_queue) ;

    stwd_end_interrupt_section(irq_num, true) ;
//END INTERRUPT SECTION

    return copy_out_svc_result(output, sizeof(query_output), &query_output) ;
}
```

Dispatching UART Requests

Following the established pattern, a device realm dispatch function for UART requests is constructed.

```
<<dev realm uart proxy: external declarations>>=
extern int
dev_realm_uart(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error) ;
```

```
<<dev realm uart proxy: external function definitions>>=
int
dev_realm_uart(
    SVC_DevRequest req,
    SVC_RequestParam const *const input,
    SVC_RequestParam *const output,
    SVC_RequestParam *const error)
{
    int status = dev_req_validate(req, uart_operation_count, UART_INSTANCES) ;
    rtcheck_zero_return(status, status) ;
```

```
    static SVC_DevRequestProxy const uart_proxies[uart_operation_count] = {
        [uart_open] = dev_realm_uart_open,
        [uart_close] = dev_realm_uart_close,
        [uart_transmit] = dev_realm_uart_transmit,
        [uart_receive] = dev_realm_uart_receive,
        [uart_query] = dev_realm_uart_query,
    } ;

    SVC_DevOperation uart_operation = dev_req_extract_operation(req) ;
    return uart_proxies[uart_operation](req, input, output, error) ;
}
```

```
<<dev realm param: constants>>=
#define DEV_UART_CLASS        4
```

```
<<svc entry: device request classes>>=
[DEV_UART_CLASS] = dev_realm_uart,
```

Privileged Transmission

There are times during privileged execution when UART output needs to be generated. For example, the `panic` function should print the panic message if the UART is being used for console output. Privileged execution cannot invoke `dev_uart_transmit` since that would cause a Hard Fault. The functions in this section are a "back door" for privileged code to get immediate access to UART output. Since output by privileged code is usually on the path to a system reset, these functions wait for all the output to be transmitted.

```
<<dev realm uart proxy: external declarations>>=
extern int
write_priv(
    int inst,
    char const *buf,
    size_t len) ;
```

inst
: The UART instance number to use for the output.

buf
: A pointer to the data to be transmitted.

len
: The number of bytes pointed to by `buf`.

The `write_priv` function writes `len` number of bytes pointed to by `buf` to the UART given by `inst`. The return value is the number of bytes written or -1 if an error occurs. The function does not return until the UART transmitter is idle.

```
<<dev realm uart proxy: external function definitions>>=
int
write_priv(
    int inst,
    char const *buf,
    size_t len)
{
    rtcheck_max_return(inst, UART_INSTANCES, -1) ;
    rtcheck_not_NULL_return(buf, -1) ;

    UART_ControlBlock *const ucb = &uart_control_blocks[inst] ;
```

```
    UART_TxQueue *const tx_queue = &ucb->tx_control.tx_queue ;
    UART0_Type *const uart = ucb->access->uart ;

    IRQn_Type irq_num = ucb->access->irqn ;
    size_t count = 0 ;

    (void)stwd_begin_interrupt_section(irq_num) ;                    // ❶
//BEGIN INTERRUPT SECTION

    while (count < len && ioqueue_insert(tx_queue, buf[count])) {
        count++ ;
    }
    (void)fill_tx_fifo(ucb) ;

    stwd_end_interrupt_section(irq_num, true) ;
//END INTERRUPT SECTION

    while (!tx_fifo_empty(uart)) {
        // empty
    }
    while (tx_busy(uart)) {
        // empty
    }

    return (int)count ;
}
```

❶ The interrupt must be disabled because the TX queue is shared data.

```
<<dev realm uart proxy: external declarations>>=
extern int
puts_priv(
    char const *s) ;
```

s
A pointer to a `NUL` terminated character string to output. A newline character is appended to the output.

The `puts_priv` function outputs the string pointed to by `s`, appending a newline character, to the primary console UART. The return value is the number of bytes output or -1 on error.

```
<<dev realm uart proxy: external function definitions>>=
int
puts_priv(
    char const *s)
{
    int written = write_priv(CONSOLE_UART, s, strlen(s)) ;
    written += write_priv(CONSOLE_UART, "\r\n", 1) ;
    return written ;
}
```

```
<<dev realm uart proxy: external declarations>>=
extern int
printf_priv(
    char const *format,
    ...) ;
```

format
: A pointer to a `NUL` terminated character string containing the format specification.

...
: A variable number of arguments which correspond to the fields of the `format` string.

The `printf_priv` function formats and outputs a string according to the `printf(3)` standard library function.

```
<<dev realm uart proxy: external function definitions>>=
int
printf_priv(
    char const *format,
    ...)
{
    char buf[256] ;

    va_list ap ;
    va_start(ap, format) ;
    int len = vsnprintf(buf, sizeof(buf), format, ap) ;
    va_end(ap) ;

    return write_priv(CONSOLE_UART, buf, len) ;
}
```

Code Layout

UART service requests

```
<<dev_svc_uart_req.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Function prototypes for UART service requests.
 *--
 */
#ifndef DEV_SVC_UART_REQ_H_
#define DEV_SVC_UART_REQ_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "dev_svc_req.h"
#include "dev_realm_uart_param.h"
```

```
/*
 * Constants
 */
<<dev svc uart req: constants>>
/*
 * Data Type Declarations
 */
<<dev svc uart req: data type declarations>>
/*
 * External Declarations
 */
<<dev svc uart req: external declarations>>

#endif /* DEV_SVC_UART_REQ_H_ */
```

UART Device Service Requests

```
<<dev_svc_uart_req.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Device Realm UART Request Implementation
 *--
 */

/*
 * Include files
 */
#include <assert.h>
#include "useful.h"
#include "rtcheck.h"
#include "svc_req_errors.h"
#include "svc_req.h"
#include "dev_svc_req.h"
#include "dev_svc_uart_req.h"
#include "dev_realm_uart_param.h"
#include "sys_svc_req.h"
/*
 * Static Function Definitions
 */
<<dev svc uart req: static function definitions>>
/*
 * External Functions
 */
<<dev svc uart req: external definitions>>
```

Device Realm UART Parameters

```
<<dev_realm_uart_param.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
```

```
 * Project:
 *   Code to Models
 *
 * Module:
 *   Parameter interface data structures for UART device requests.
 *--
 */
#ifndef DEV_REALM_UART_PARAM_H_
#define DEV_REALM_UART_PARAM_H_
/*
 * Include Files
 */
#include "dev_realm_param.h"
#include "apollo3.h"
/*
 * Constants
 */
<<dev realm uart param: constants>>
/*
 * Static Inline Functions
 */
<<dev realm uart param: static inline functions>>
/*
 * Data Type Declarations
 */
<<dev realm uart param: data type declarations>>

#endif /* DEV_REALM_UART_PARAM_H_ */
```

Device Realm UART Proxy Header

```
<<dev_realm_uart_proxy.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Interfaces for device realm UART proxy functions.
 *--
 */
#ifndef DEV_REALM_UART_PROXY_H_
#define DEV_REALM_UART_PROXY_H_

/*
 * Include files
 */
#include "dev_realm_param.h"
/*
 * External Declarations
 */
<<dev realm uart proxy: external declarations>>

#endif /* DEV_REALM_UART_PROXY_H_ */
```

```
<<svc handler: device include files>>=
#include "dev_realm_uart_proxy.h"
```

Device Realm UART Proxy Code Layout

```
<<dev_realm_uart_proxy.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Implementation for device realm UART proxy functions.
 *--
 */

/*
 * Include files
 */
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include "svc_req_errors.h"
#include "svc_proxy.h"
#include "dev_realm_proxy.h"
#include "dev_realm_gpio_param.h"
#include "dev_realm_gpio_proxy.h"
#include "dev_realm_uart_param.h"
#include "dev_realm_uart_proxy.h"
#include "bg_req_queue.h"
#include "panic.h"
#include "bit_twiddle.h"
#include "sys_twiddle.h"
#include "dev_twiddle.h"
#include "useful.h"
#include "rtcheck.h"
<<dev realm uart proxy: include files>>
/*
 * Constants
 */
<<dev realm uart proxy: constants>>
/*
 * Macros
 */
<<dev realm uart proxy: macros>>
/*
 * Data Type Declarations
 */
struct uart_controlblock ;                              // forward reference
<<dev realm uart proxy: data type declarations>>
<<dev realm uart proxy: io pin data type declaration>>
<<dev realm uart proxy: uart access data type declaration>>
<<dev realm uart proxy: io queue type declaration>>
<<dev realm uart proxy: framer type declarations>>
<<dev realm uart proxy: uart tx control type declaration>>
<<dev realm uart proxy: uart rx control type declaration>>
<<dev realm uart proxy: uart control block type declaration>>
/*
 * Forward References
 */
<<dev realm uart proxy: forward references>>
```

```
/*
 * Static Data Definitions
 */
<<dev realm uart proxy: static data definitions>>
<<dev realm uart proxy: uart instances definitions>>
/*
 * Static Inline Function Definitions
 */
<<dev realm uart proxy: static inline functions>>
/*
 * Static Function Definitions
 */
<<dev realm uart proxy: static function definitions>>
/*
 * External Function Definitions
 */
<<dev realm uart proxy: external function definitions>>
```

Console I/O

With the UART device code in place, better handling of console output is possible. Until now, SEGGER RTT has been used as the I/O mechanism. This is convenient to implement, but has a number of drawbacks.

- RTT requires an attached JLink connection.
- Low power mode interacts poorly with RTT, forcing us to flush output before entering deep sleep.

To overcome these problems, this section shows the implementation of `printf` style formatted console I/O using the UART device. This code will allow building simpler stand alone applications.

When using the console output, there is an implicit assumption that no other background processing is using making direct calls to the `dev_uart_...` request functions. This is reasonable for the Apollo 3 peripherals which are oriented to SPI and I^2C.

As discussed previously, UART 0 in the Sparkfun MicroMod design is effectively dedicated to bootloader operations. That leaves us with UART 1 to use for console output. UART 1 also supports hardware flow control which is needed to run at high baud rates with minimal buffering.

```
<<dev realm uart param: constants>>=
#ifndef CONSOLE_UART
#   define CONSOLE_UART        1U
#endif /* CONSOLE_UART */
```

```
<<dev realm uart param: constants>>=
#ifndef CONSOLE_BAUD_RATE
#   define CONSOLE_BAUD_RATE   921600U
#endif /* CONSOLE_BAUD_RATE */
```

Design Overview

There are two design goals to meet:

- To re-target standard library input and output functions from `newlib` to use the UART device.
- To supply a `printf` function which uses the UART device for output.

The strategy for meeting these goals is:

- Supply a replacement function for the system `_write` function.
- Supply a replacement function for the system `_read` function.
- Supply a replacement `printf` function by using `snprintf` to place the formatted output into a local variable. One reason for supplying `printf` is that the newlib version of the function uses `malloc` to obtain memory and dynamic allocation from a system wide heap is not supported in this design.
- Use `sys_ctrl_busy_wait` in conjunction with a background notification proxy to synchronize simulate synchronous operation.

Console I/O Background Notification Proxy

A background notification proxy is supplied that handles the three types of UART notifications by writing to a separate variable for each notification type.

```
<<console output: uart notification>>=
static bool volatile uart_tx_notified ;
static bool volatile uart_rx_notified ;
static bool volatile uart_break_notified ;
```

```
<<console output: uart notification>>=
static void
uart_notify_proxy(
    SVC_DevUartNotification const *const notification)
{
    switch (notification->type) {
    case uart_notify_tx_available:
        uart_tx_notified = true ;
        break ;

    case uart_notify_rx_available:
        uart_rx_notified = true ;
        break ;

    case uart_notify_break:
        uart_break_notified = true ;
        break ;
    }
}
```

Functions for the background processing to use for synchronization when the UART is being handled as a console I/O device are provided.

```
<<dev svc uart req: external declarations>>=
extern void console_rx_sync(void) ;
extern void console_tx_sync(void) ;
extern void console_break_sync(void) ;
```

```
<<console output: external function definitions>>=
void
console_rx_sync(void)
{
    sys_ctrl_busy_wait(&uart_rx_notified) ;
}

void
console_tx_sync(void)
{
    sys_ctrl_busy_wait(&uart_tx_notified) ;
```

```
}

void
console_break_sync(void)
{
    sys_ctrl_busy_wait(&uart_break_notified) ;
}
```

`_write` Function Replacement

The `_write` function is where all output from `newlib` is routed.

```
<<console output: external function definitions>>=
#ifdef USE_UART

int
_write(
    int file,
    void const *ptr,
    size_t len)
{
    (void)file ;                    // not used
    rtcheck_not_NULL_return(ptr, 0) ;
    rtcheck_not_zero_return(len, 0) ;

    console_io_init(uart_frame_terminal) ;

    uint8_t const *src = ptr ;
    int count = 0 ;
    while (count < len) {
        int transmitted = dev_uart_transmit(CONSOLE_UART, len, src) ;

        if (transmitted < 0) {
            count = -1 ;
            break ;
        } else {    // transmitted >= 0
            len -= transmitted ;
            src += transmitted ;
            count += transmitted ;
        }
    }

    return count ;
}

#endif /* USE_UART */
```

The UART must be opened before it can be written to. This initialization function is designed to be invoked multiple times without bad effect.

```
<<dev svc uart req: external declarations>>=
extern void console_io_init(UART_FramingType framing) ;
```

```
<<console output: external function definitions>>=
void
console_io_init(
    UART_FramingType framing)
{
    static bool uart_initialized = false ;
```

```
    if (!uart_initialized) {
        int status = dev_uart_open(CONSOLE_UART, CONSOLE_BAUD_RATE, true, framing,
                uart_notify_proxy, 0) ;
        assert(status == 0) ; (void)status ;
        uart_initialized = true ;
    }
}
```

`_read` Function Replacement

The `_read` function is where all input to `newlib` is obtained.

```
<<console output: external function definitions>>=
#ifdef USE_UART

int
_read(
    int file,
    void *ptr,
    size_t len)
{
    (void)file ;                    // not used
    rtcheck_not_NULL_return(ptr, 0) ;
    rtcheck_not_zero_return(len, 0) ;

    console_io_init(uart_frame_terminal) ;

    uint8_t *dst = ptr ;
    int count = 0 ;
    while (count < len) {
        int received = dev_uart_receive(CONSOLE_UART, len, dst, NULL) ;

        if (received < 0) {
            count = -1 ;
            break ;
        } else if (received == 0) {
            console_rx_sync() ;
        } else {    // received > 0
            count += received ;
            if (memchr(dst, '\n', received) != NULL) {
                break ;
            }

            len -= received ;
            dst += received ;
        }
    }

    return count ;
}

#endif /* USE_UART */
```

`printf` Function Replacement

Our strategy for implementing a replacement `printf` is to use `vsnprintf` to format into a local buffer which is then written via `_write`.

The size of the local buffer is set to 256 bytes and is deemed sufficient. The return value of `vsnprintf` indicates if overflow happened. If overflow has occurred, only the amount placed in the buffer is written. Any overflow is silently truncated.

```
<<console output: constants>>=
#ifndef MAX_PRINTF_OUTPUT
#   define  MAX_PRINTF_OUTPUT   256
#endif /* MAX_PRINTF_OUTPUT */
```

```
<<console output: external function definitions>>=
#ifdef USE_UART

__attribute__((format(printf, 1, 2)))
int
printf(
    char const *format,
    ...)
{
    char buf[MAX_PRINTF_OUTPUT] ;

    va_list ap ;
    va_start(ap, format) ;
    int len = vsnprintf(buf, sizeof(buf), format, ap) ;
    int towrite = len >= sizeof(buf) ? sizeof(buf) - 1 : len ;  // ❶
    va_end(ap) ;

    return _write(0, buf, towrite) ;
}

#endif /* USE_UART */
```

❶ "- 1" because `vsnprintf` always `NUL` terminates the output string.

Code Layout

```
<<console_io.c>>=
<<edit warning>>
<<copyright info>>

/*
 * Include files
 */
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <assert.h>
#include <stdarg.h>
#include "rtcheck.h"
#include "bip_buffer.h"
#include "sys_svc_req.h"
#include "svc_req_errors.h"
#include "dev_svc_uart_req.h"
/*
 * Constants
 */
<<console output: constants>>
/*
 * Data Type Declarations
 */
<<console output: data type declarations>>
```

```
/*
 * Forward References
 */
<<console output: forward references>>
/*
 * Static Data Definitions
 */
<<console output: static data definitions>>
/*
 * Static Inline Function Definitions
 */
<<console output: static inline functions>>
/*
 * Static Function Definitions
 */
<<console output: uart notification>>
<<console output: static function definitions>>
/*
 * External Function Definitions
 */
<<console output: external function definitions>>
```

Echo Example

This example shows an echo server working with raw input. Any input received is immediately transmitted. This example demonstrates reliable communications with only a relatively small buffer and hardware flow of control.

```
<<echo server: main>>=
void
main(void)
{
    console_io_init(uart_frame_identity) ;

    uint8_t buf[128] ;
    UART_RxErrStatus err_status ;

    for (;;) {
        int rx_count = dev_uart_receive(CONSOLE_UART, sizeof(buf), buf,
                &err_status) ;
        rtcheck(rx_count >= 0) ;
        while (rx_count == 0) {
            console_rx_sync() ;
            rx_count = dev_uart_receive(CONSOLE_UART, sizeof(buf), buf,
                    &err_status) ;
            rtcheck(rx_count >= 0) ;
        }
        if (err_status != uart_success) {
            printf("receive error = %u, count = %d\n", err_status, rx_count) ;
        }

        uint8_t *pbuf = buf ;
        int tx_count = dev_uart_transmit(CONSOLE_UART, rx_count, pbuf) ;
        rtcheck(tx_count >= 0) ;
        while (tx_count < rx_count) {
            console_tx_sync() ;

            pbuf += tx_count ;
            rx_count -= tx_count ;
```

```
            tx_count = dev_uart_transmit(CONSOLE_UART, rx_count, pbuf) ;
            rtcheck(tx_count >= 0) ;
        }
    }

    int status = dev_uart_close(CONSOLE_UART) ;
    assert(status == 0) ; (void)status ;
}
```

Code Layout

```
<<speak-to-me-echo-test.c>>=
<<edit warning>>
<<copyright info>>

#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include <string.h>
#include "rtcheck.h"
#include "sys_svc_req.h"
#include "dev_svc_uart_req.h"

<<echo server: main>>
```

Console I/O Example

This example demonstrates an echo server using standard library functions and configuring the UART for interactive framing.

```
<<speak-to-me-console-test.c>>=
<<edit warning>>
<<copyright info>>

#include <stdio.h>
#include <stdbool.h>
#include "sys_svc_req.h"
#include "dev_svc_uart_req.h"

void
main(void)
{
    console_io_init(uart_frame_interactive) ;

    sys_util_panic_msg_print() ;

    char buf[128] ;

    while (fgets(buf, sizeof(buf), stdin) != NULL) {
        fputs(buf, stdout) ;
    }
}
```

Chapter 10

Conclusion

This is the end of Part I and it is appropriate to review what has been covered. There are three primary subjects in Part I.

1. Foundational systems programming for the ARM Cortex-M4 core.
2. Design of a mechanism to support the concurrency associated with handling the system environment.
3. System specific programming to incorporate the Apollo 3 peripheral devices.

Systems Programming of the Core

The Cortex-M4 core, like any other processor core, requires specific code to use it capabilities. As microcontroller cores go, the Cortex-M4 is quite capable, providing several useful features which can be used to design more robust execution environments. Compared with older 8-bit and 16-bit microcontrollers, the v7-M architecture brings capability usually found in general purpose computers. The design of the core determines those capabilities and several choices were made as to how the core is operated. Specifically:

Dual stacks
: Two stacks were used to have better flexibility in allocating stack memory. A two stack arrangement makes it easier to handle the required stack space for higher priority exceptions and for those exceptions whose priority is fixed. The Hard Fault and NMI exceptions do not have configurable priorities and the fixed priority assigned to them is higher than the priority of other configurable exceptions. This means that there must always be sufficient space on any stack to accommodate handling those exceptions. By separating the Main Stack from the Process Stack, the Process Stack does not need to account for the stack usage that might happen as a result of preemption of one exception by another. Once one first exception is taken, any subsequent exception that runs as a result of preemption uses the Main Stack. The Process Stack then only needs to be sized to handle the requirements of the application plus the space required to stack the core context when any exception is taken. Sizing the Main Stack can be tailored to the exception priority scheme used. In this design, the priority scheme has only four possibilities for preemption: fixed priority system faults, configurable priority system faults, debug monitor, and interrupt requests. The two stack design also interacts with the use of memory protection insuring that unprivileged code does not have access to the Main Stack.

Privileged execution
: Execution privilege was separated so that application logic runs in an unprivileged manner. This separation is essential for general purpose operating systems and beneficial for managing the run time of a microcontroller. This is in keeping with the emphasis on separating system level concerns. Interactions with hardware peripherals and the external environment are considered privileged. The intent is to be able to build well honed and tested privileged code which changes little. Application logic varies as the primary purpose of the system and is subject to more frequent changes.

Memory protection unit
: The MPU is used to enforce access to memory based on the separation of execution privilege. The scheme used in this book is simple and directly based on the execution privilege and the use of the memory by the running program. This accounts for the difference in instructions and data as well as the difference between flash memory and RAM.

System Environment Concurrency

A goal of the design in Part I was to manage the concurrency of the environment using the capabilities of the core and a minimum of additional software. The central concept is to use a notification queue as the mechanism to transport data and control into the application processing.

The separation of privileged and unprivileged execution necessitated the use of the `SVC` instruction to allow unprivileged code to request privileged operations. Those requests run to completion before returning to the requesting background code.

Concurrent interactions with the system environment are represented by interrupt requests. Handling an IRQ can result in additional computing that needs to be done in the background. Deferring execution to background code shortens the execution time of the IRQ handler and so lowers the latency of responding to another interrupt. The Background Notification Queue is the mechanism where notifications are sent to the background for further action. Because the timing of arrival of interrupt requests is often not predicable, the queue serves as a buffer, allowing the IRQ handling and the subsequent background processing to be disconnected in time from each other.

The particulars of the operation of the notification queue allow flexibility in the size and contents of the notification messages. The queue is the only mechanism provided or needed for foreground processing to interact with background processing. In particular, *ad hoc* methods with shared global data between foreground and background are unavailable because of execution privilege separation. This necessitated a design of the notification queue that is completely independent of the subject matter of the application.

Peripheral Device Operations

Since microcontroller-based reactive systems depend heavily on interacting with the system environment through their peripheral devices, several chapters were devoted to demonstrating how the concurrency scheme embodied in the Background Notification Queue could be used to present logical devices for use by application software.

The first examples shown, a watchdog timer and a timing queue, were based on timer peripherals as a matter of convenience. Timers used for generating a stimulus do not have physical connections to the outside world. They simply count the ticks in the "ether."

Connections to the outside world are important, so control of simple GPIO pins was also demonstrated. Finally, UART communications was demonstrated since it is a basic and essential tool for microcontroller software development.

In future parts of the book, additional peripheral device control will be shown. Device control always makes up a substantial part of a microcontroller system. Such code is often more difficult to produce since it involves not only complex logic of the peripheral device but often considerations of timing and scale must be accommodated. Note that there has been no attempt to create a Hardware Abstraction Layer, *per se*. This is because no attempt has been made to support peripheral device functionality that did not have an immediate use. There are clear places in the design where the direct operations on a peripheral device were factored out. But there is no concerted effort to make every possible configuration of a peripheral accessible from a function in the code. This is an effort to create a running system, not a software development kit (SDK).

Part II

Baton

N.B. the single chapter included in this part of the book is a placeholder. It represents a port of the `micca` run time to operate on the notification queue mechanism defined in Part I. Part II of this book is incomplete, but the included chapter gives much of the substance that will eventually be covered.

Chapter 11

Runtime Support

In the previous part of the book, we described the domain specific language (DSL) that is used to describe a domain to `micca`. From that description, code is generated and the generated code targets the run-time environment that is explained in this part of the book.

When translating an executable model, parts of the model can be mapped directly onto the implementation language. For example in a `micca` translation, class instances are referred to using a "C" pointer to the instance memory. This allows attribute access by simple indirection on the pointer, *e.g.* for a state activity, `self->Temp = 27`, can be used to update the `Temp` attribute of the instance.

Other aspects of mapping model execution to the implementation are not directly supported in the implementation language. Consider the execution of a state model. There is no intrinsic support for state machines in the "C" language. To dispatch events to state machines requires that we write additional "C" code to implement the execution rules of Moore state machines. We will gather that code into "C" functions and the functions will require data that must be structured in a particular way.

This part of the book is primarily concerned with showing the functions and data structures required to map model execution rules onto the implementation. Taken together, they define the details of how execution happens in a `micca` translated domain. We have purposely chosen a simple scheme of execution, namely, single threaded and event driven with callbacks to handle the domain specific computations. This choice enables this translation scheme to be used in small, embedded, "bare metal" computing technology. We will also produce a flavor of the run-time that can execute in a POSIX environment. This will allow us to target POSIX, if that is appropriate to the application, and also to use POSIX as a simulation environment for testing an embedded system. This turns out to be useful since debugging and other facilities tend to be much richer in the POSIX world than in the bare metal embedded world.

Once we know how the functions and data structures of the run-time code operate, we can show the code generation that takes the platform model data and converts it to "C" code. That is the subject of the next part of the book.

Introduction to the Micca Run Time

To start, we enumerate the characteristics of the computing technology targeted by the `micca` run-time.

- The implementation language is "C".
- All data is held in primary memory and no automatic persistence of data to external storage is provided.
- All resources which require dynamic memory allocation perform the allocation from fixed size memory pools whose size is known at compile time. The run time code does not perform any dynamic memory allocation from a system heap (*e.g.* no calls to `malloc`).
- The address in memory of a class instance serves as an architecturally generated identifier for the instance and we use that identifier exclusively in the interfaces of the run-time code.

- No other constraints on the identification of class instances is provided, particularly, no enforcement of uniqueness based on attribute values is provided by the run-time. Translations are strongly encouraged to provide such enforcement, especially when the identifying attribute values are obtained from outside of a domain.
- Background execution is single threaded. There is no notion of *task* or *process* or *thread* as those terms are usually defined for computer operating systems. There is no notion of processor context, context switching or process preemption.
- The runtime assumes the existence of an interface to the environment of the system which handles interrupts and the flow of data across the system boundary. There is no notion of semaphores, condition variables or any other mutual exclusion locking scheme. All background execution is run-to-completion with interrupts enabled.
- The execution pattern is event driven with callbacks that run to completion and perform the required computation. Given the single threaded nature of sequencing execution, callbacks that run for longer than the desired response latency time of the system can be problematic. This scheme is targeted at applications that are *reactive* in nature, sensing external stimulus and responding to that stimulus, and *not* long-running computationally intensive transformations. The execution scheme is similar to that used for most graphical user interface applications.
- There is no notion of event priority. Events are dispatched in the order they are signaled[1].
- Events are dispatched within the context of a *thread of control*. Threads of control have an direct corollation with transactions on the domain data.
- Referential integrity is enforced at the end of each data transaction. This implies that any changes in the stored data for relationships is verified against the constraints implied by the relationship conditionality and multiplicity. Referential integrity is evaluated whenever there has been any change in class instances or relationship instances during a thread of control.

As we have said before, this execution scheme is *not* intended for all applications. The computational demands of some applications are such that the single threaded, event driven, callback nature is simply inappropriate and another execution scheme should be used. However, for a large class of applications, this scheme works well and has many advantages in terms of simplicity of the implementation and the absence of data races for shared information between concurrent execution threads.

Limitation of the Run Time

Although `micca` supplies everything needed to sequence execution and manage model data, there are several other components of a target platform that are **not** supplied by `micca`. Notably:

- Exception handling is system specific. The details of how processor exceptions are handled must be supplied by the project. The run-time provides the means to synchronize between exceptions and the background processing but nothing else.
- The run-time requires access to some type of timing resource. The interface to the timing resource is well defined but projects have to supply code to manipulate the timing resource. Typically in an embedded context, a hardware timer is used.

Conditional Compilation

The run-time support the following “C” preprocessor symbols:

Important
It is important the code files for the run-time and all the domains that constitute an application be compiled with the same set of preprocessor symbol definitions.

`NDEBUG`
The run-time uses the standard `assert` macro and the assertions may be removed by defining this symbol.

[1] With the exception that self directed events are dispatched before non-self directed ones. However, this is one of the prescribed execution rules of state models.

MRT_NO_NAMES
If defined, this symbol will exclude naming information about classes, relationships and other domain entities from being compiled in. For small memory systems, strings can consume a considerable amount of space and are usually only used during debugging.

MRT_NO_TRACE
Defining this symbol removes code from the run-time that traces event dispatch. Event dispatch tracing is important during testing and debugging but may be removed from the delivered system.

MRT_NO_STDIO
Defining this symbol insures that `stdio.h` is not included and no references are made to functions in the standard I/O library. This is useful for smaller embedded systems that cannot support the memory required by the standard I/O library.

MRT_TRANSACTION_SIZE
The value of this symbol sets the maximum number of relationships that can be modified during a data transaction. The default value is 64.

MRT_EVENT_POOL_SIZE
The value of this symbol sets the number event control blocks which are used for signaling events. The default value is 32. This number represents the maximum number of signaled events that may be *in flight* at the same time.

MRT_ECB_PARAM_SIZE
This value of this symbol set the maximum number of bytes that can be occupied by event parameters or sync function parameters. The default value is 32.

MRT_INSTANCE_SET_SIZE
The value of this symbol is the maximum number of instance references that may be held in an instance reference set. The default value is 128.

MRT_INSTRUMENT
Defining this symbol includes code to print the function name, file name and line number for all functions generated by the code generator. This information forms a trace of executed functions.

MRT_INSTRUMENT_ENTRY
Defining this symbol can override the code that is placed at the beginning of each generated function for instrumentation. By default when `MRT_INSTRUMENT` is defined the following code is placed at the entry of each generated function:

```
printf("%s: %s %d\n", __func__, __FILE__, __LINE__) ;
```

MRT_DEBUG(...)
The `MRT_DEBUG` macro has the same invocation interface as `printf()`. If `MRT_INSTRUMENT` is defined, then `MRT_DEBUG` invocations will include the implied `printf` invocations. Otherwise, the implied `printf` invocations are removed from the code (*i.e.* `MRT_DEBUG` is defined as empty). If `MRT_INSTRUMENT` is defined, then the expansion of `MRT_DEBUG` may be overridden.

The Main Program

In “C”, execution begins with the function called, `main`. This function is *not provided* by the runtime. Each application has to supply a suitable `main` function. The goals of `main` are to perform any required initialization and then to enter the event loop. Below we show an outline of what a `main` function would do.

```
int
main(
    int argc,
    char *argv[])
{
    /*
```

```
     * Hardware and other low level system initialization is usually done
     * first.
     */

    /*
     * Initialize the run-time code itself.
     */
    mrt_Initialize() ;

    /*
     * Initialize domains, bridges and any other code that might require access
     * to the facilities of the run-time code. Typically, each domain in the
     * system would have an "init()" domain operation and these can be invoked
     * here. Sometimes domain interactions are such that a second round of
     * initialization is required.  Bridges between domains may also require
     * that the initialization for a domain be done before the bridge can be
     * initialized. Once mrt_Initialize() has been invoked, domains may
     * generate events and do other model level activities.  Regardless of how
     * the initialization is accomplished, it is system specific and,
     * unfortunately, only temporally cohesive.
     */

    /*
     * Entering the event loop causes the system to run.
     */
    mrt_EventLoop() ;

    /*
     * It is possible that domain activities can cause the main loop to exit.
     * Here we consider that successful. Other actions are possible and
     * especially if the event loop is exited as a result of some unrecoverable
     * system error.
     */
    return EXIT_SUCCESS ;
}
```

The only hard and fast requirements are that `mrt_Initialize` must be called before any facilities of the run-time are used.

```
<<mrt external interfaces>>=
extern void
mrt_Initialize(void) ;
```

The `mrt_Initialize` function initializes the micca run-time. This function should be called early in the initialization of a system and must be invoked before using any run-time facilities such as signaling an event.

The implementation of `mrt_Initialize` is target platform specific. So we postpone showing its code until later when we discuss how the run-time is tailored to specific target computing platforms.

```
<<mrt external functions>>=
void
mrt_Initialize(void)
{
    mrtECBPoolInit() ;
    mrtTransactionsInit() ;
    mrtInitSysTimer() ;
#       if !(defined(MRT_NO_TRACE) || defined(MRT_NO_STDIO))
    mrt_RegisterTraceHandler(mrtPrintTraceInfo) ;
#       endif /* !defined(MRT_NO_TRACE) && !defined(MRT_NO_STDIO) */
}
```

```
<<mrt implementation constants>>=
#define MRT_TIMER_QUEUE_DEVICE   0
```

```
<<mrt static functions>>=
static void
mrtInitSysTimer(void)
{
    int status = dev_timq_open(MRT_TIMER_QUEUE_DEVICE, mrtExpireDelayedEvent) ;
    assert(status == 0) ;
    if (status < 0) {
        mrtFatalError(mrtTimerOpFailed, "open") ;
    }
}
```

Runtime Use Cases

The `micca` runtime supports three use cases for causing the system to execute.

1. An infinite event loop.

   This is the primary intended mode of operation. Once the system is initialized, the event loop is entered and execution remains there forever.

2. Single thread of control.

   This use is intended to help integration of `micca` based code into an existing execution control scheme. This case comes up, for example, when a system is transitioned to being `micca` generated but there is existing code which controls execution sequencing. An existing event loop can be modified to periodically execute `micca` threads of control to coordinate the different execution schemes. This case is also useful for debuggers and testing. Executing a thread of control corresponds, roughly, to a *next* command in a conventional code debugger. A thread of control also roughly corresponds to a *unit* for testing purposes. Executing a complete thread of control insures that a data transaction on the domain is completed and the domain population is in a consistent state. The `bosal` test harness generator uses this mode extensively.

3. Single event.

   This use case is intended almost exclusively for debugging and simulation. Executing a single event is analogous to a *step* command in a conventional code debugger. Executing a single event may leave a domain population in an inconsistent state in the case where a thread of control consists of dispatching multiple events.

There is a fourth situation for controlling execution which arises in conjunction with error handling. As described in the following error handling section, it is possible to install an application specific error handler which is executed when an error occurs. An application specific error handler may use `setjmp/longjmp` to jump back to `main` when an error occurs. By default, all runtime errors are *fatal* in that they end up calling `abort()`. Such default behavior is not desired in many situations and the use of `setjmp/longjmp` is a standard mechanism to handle errors detected at lower levels in the call tree. The runtime supports this usage by insuring that resuming event dispatch can continue if the error which the problem was corrected. This use case is also important for both integrating with other event loops and for testing purposes.

The following sections discuss the external functions provided to support the three use cases.

Event Loop

The figure below shows a simplified diagram of the flow of control in the event loop.

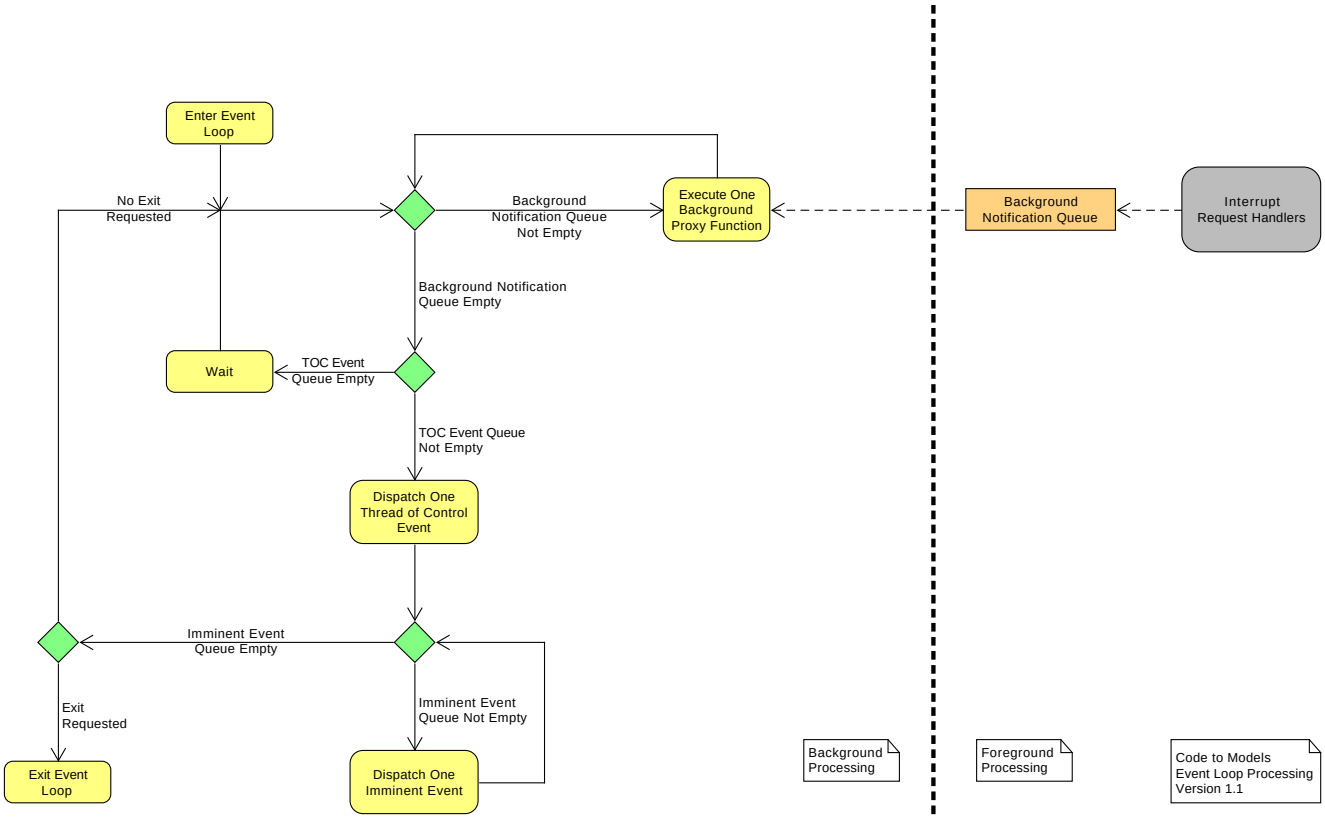

Figure 11.1: Micca Run Time Event Loop

Processing is divided between the background and foreground. Foreground processing happens as a result of an interrupt[2]. A queue is used to buffer and synchronize between the two execution realms. The background processing is single threaded. We will discuss the internal logic of processing an event later. For now, we make these points.

- If interrupts have occurred, we deal with their background synchronization first and completely. The exact manner used to synchronize between interrupts and the background is processor architecture dependent.
- Any event signaled by a sync function starts a thread of control. Only one thread of control event is dispatched after synchronizing with interrupts. Because we invoke the synchronization functions requested by all the interrupts before dispatching any event, interrupt synchronization has an effective higher priority than event dispatch.
- If the event queue becomes empty, we check if any activity has requested that the event loop terminate. If so, then flow continues to the remainder of the `main` program.
- If the event loop is not exited, the run-time waits until there is further work to be done. Note that any further work must arise from interrupt service and its synchronization to the background. Waiting is platform specific, but usually, for embedded systems, involves going into a sleep or low power mode. For the POSIX version of the run-time, the `pselect(2)` system call is used to suspend the process awaiting either a signal or a change in status of a file descriptor.

As we shall see below, the above description of the event loop flow is somewhat simplified. There is one additional concept for which we must account. As stated previously, events originating outside of an instance context executing an activity start a new *thread of control*. This includes events that originate from a domain operation or are delayed events. Delayed events (even if self directed) are considered to be delivered by the run-time and hence outside of a state activity.

[2]In the POSIX environment, signals and changes in state of file descriptors serve the same role as an interrupt

The thread of control terminates when the event that started it and all the other events that arose from the starting event have been dispatched. The event that starts a thread of control causes a transition that runs a state activity. That activity may signal other events, causing other transitions which may signal yet more events. At some point, the tree of state activity invocations does not produce any further events. When the entire set of events is dispatched, the thread of control ends. When a thread of control ends, the referential integrity of the data is checked. There is a close association between a transaction on the domain data and the bounds of the thread of control. At the boundary of threads of control, run-time evaluates requests to exit the event loop. The event loop will not normally be exited until the end of any ongoing thread of control

To keep the event dispatches that start a thread of control separate from those that run during a thread of control, the run-time keeps two queues. All events generated outside of a state machine activity are queued to the thread-of-control queued while those generated by state machine activities are queued to an immediate dispatch event queue. We will see those queues in the code below,

```
<<mrt external interfaces>>=
extern void
mrt_EventLoop(void) ;
```

The `mrt_EventLoop` function enters the micca run-time event loop and dispatches events to cause the system to run. Control remains in the event loop unless specifically requested to exit via the `mrt_SyncToEventLoop` function.

Important
The `mrt_EventLoop` function must not be called recursively, *i.e.* `mrt_EventLoop` is not to be invoked as part of an activity. The system must remain in the event loop until properly exited. Only after the event loop has been exited may it be re-entered.

To a first approximation, the event loop is an infinite loop. Most applications will enter the event loop after their initialization phase and remain there forever.

```
<<mrt external functions>>=
void
mrt_EventLoop(void)
{
    mrt_BusyLoop(&mrtExitEventLoop) ;
}
```

As we see, the main event loop is just a base level busy loop which processes one thread of control at a time as long as it has not been requested to exit the loop.

Testing and other considerations mean that we may want to exit the event loop to gain control of the program. Exiting the event loop is controlled by a boolean variable.

```
<<mrt static data>>=
static bool mrtExitEventLoop ;
```

Rather than expose the variable directly, we provide a function to set it.

```
<<mrt external interfaces>>=
extern bool
mrt_SyncToEventLoop(void) ;
```

The `mrt_SyncToEventLoop` function will cause the event loop to exit after completing any ongoing thread of control. The function returns `false` if the event loop would not have exited and `true` otherwise, *i.e.* the function returns the exit control value before it was modified.

One use case for this function is in test code where the main test application will enter the event loop and some state activity will execute `mrt_SyncToEventLoop()` to return control back to the test application. There are other uses. For example, if the

system detects a catastrophic situation, it may want to exit the event loop and allow the subsequent code to gracefully bring down the system or force the system to reset. The exact details are system specific, but this provides the means to gain control of the run-time execution for system specific circumstances.

```
<<mrt external functions>>=
bool
mrt_SyncToEventLoop(void)
{
    bool exitControl = mrtExitEventLoop ;
    mrtExitEventLoop = true ;
    return exitControl ;
}
```

The bulk of the control sequencing is performed in the following function.

```
<<mrt external interfaces>>=
void
mrt_BusyLoop(
    bool *done) ;
```

done
A pointer to a boolean variable whose value determines when the busy loop exits.

The `mrt_BusyLoop` function set the boolean value pointed to by `done` to false and then repeatedly run a thread of control until the value pointed to by `done` is set to true. All asynchronous processing happens as if the event loop has been entered. Presumably, some piece of background processing sets the value pointed to by `done` to `true` to cause the busy loop to exit. This function can be used to force certain kinds of synchronous execution while still keep the asynchronous activity of the system running. The main event loop is just a special case usage of `mrt_BusyLoop`.

The `mrt_BusyLoop` function may be invoked multiple times. Such invocations nest and in most cases use different `done` variables.

```
<<mrt external functions>>=
void
mrt_BusyLoop(
    bool *done)
{
    *done = false ;

    for (;;) {
        bool didtoc = mrtRunThreadOfControl() ; // ❶

        if (*done) {                            // ❷
            break ;
        } else if (!didtoc) {                   // ❸
            sys_wait() ;
        }
        /*
         * Else we ran a thread of control and weren't requested to exit
         * the event loop.
         */
    }
}
```

❶ Run one thread of control, recording whether there was any work done.

❷ Check if we have been requested to exit the event loop. We do this first because it is possible that something took us out of `sys_wait()` and requested to exit the event loop without actually signaling an event (and so no thread of control actually ran). For example, this happens when domains are run under a test harness.

❸ If no thread of control actually ran, then wait for something in the outside world to wake us up.

The `sys_wait()` function puts the processor to sleep until an interrupt occurs. We need header files for the function prototypes.

```
<<mrt implementation includes files>>=
#include "sys_svc_req.h"
#include "svc_req_errors.h"
```

Dispatching a Thread of Control

The second mechanism provided for controlling execution is to dispatch a single thread of control. Testing and some other situations are most easily handled if we can dispatch all the events in a single thread of control, waiting for the starting event to come along if necessary.

```
<<mrt external interfaces>>=
extern bool
mrt_DispatchThreadOfControl(
    bool wait) ;
```

`wait`
If `wait` is `true`, and there is no thread of control event to dispatch, then the function blocks until one comes along and then runs the thread of control.

The `mrt_DispatchThreadOfControl` function attempts to run a single thread of control and then returns to the caller a boolean value to indicate if a thread of control was actually executed. Any ongoing thread of control is finished and is not considered in the return value. If there are no queued events that would start a thread of control, then the function waits until one is queued if the `wait` argument is `true` and then runs the thread of control that results. If `wait` is `false` and there is no thread of control ready to run, then the function returns `false` immediately. Since this function will wait for a thread of control event to arrive, it can be useful to wait for a delayed event or an event that arises from the external environment. It is possible that a foreground / background synchronization could occur that wakes up the processor but did not start a thread of control. In that case as well as when the `wait` argument is `false`, the function could return `false`.

Important
The `mrt_DispatchThreadOfControl` function must not be called recursively, *i.e.* `mrt_DispatchThreadOfControl` is not to be invoked as part of an activity. The system must remain in the event loop until properly exited. Only after the event loop has been exited may it be re-entered.

```
<<mrt external functions>>=
bool
mrt_DispatchThreadOfControl(
    bool wait)
{
    bool rantoc = mrtRunThreadOfControl() ;
    if (!rantoc && wait) {
        sys_wait() ;
        rantoc = mrtRunThreadOfControl() ;
    }
    return rantoc ;
}
```

We also provide an interface to dispatch one event and then return to the caller.

```
<<mrt external interfaces>>=
extern bool
mrt_DispatchSingleEvent(void) ;
```

Note
The `mrt_DispatchSingleEvent` function is deprecated and will be removed in a future release. The proper granularity for event dispatch is the *thread of control*.

The `mrt_DispatchSingleEvent` function may be invoked to process a single event and then returns to the caller. The return value for `mrt_DispatchSingleEvent` indicates if an event was actually dispatched. A return value of `false` indicates that the event queue is empty. This function ignores thread of control boundaries and so will dispatch the first available event even if that event starts a thread of control.

Important
The function `mrt_DispatchSingleEvent` is not to be invoked while in the event loop, *i.e.* `mrt_DispatchSingleEvent` is not to be invoked by domain activities. It is only valid to invoke this function if the program is not currently running under the control of the event loop.

This function can be used to control the dispatch of each event. This is *not* the recommended mechanism of running a `micca` based application, but sometimes it is necessary. Again, testing is sometimes best accomplished by letting a test program control event-by-event dispatch. Another use case involves situations where `micca` generated domains must be integrated with legacy code. The legacy code will probably have its own execution loop and we will have to dispatch `micca` events under control of the legacy execution scheme. This is not an ideal situation, but when migrating an older system to a `micca` translated system a transition period is unavoidable. We provide the ability to micro-manage the event dispatch with the full realization that the capablity will be abused by some projects.

```
<<mrt external functions>>=
bool
mrt_DispatchSingleEvent(void)
{
    bool dispatched = mrtEventQueueEmpty(&eventQueue) ?
            mrtDispatchTOCEvent() :
            mrtDispatchEventFromQueue(&eventQueue) ;

    if (mrtEventQueueEmpty(&eventQueue)) {                  // ❶
        mrtEndTransaction() ;
    }

    return dispatched ;
}
```

❶ We must make sure to enforce referential integrity whenever the thread of control is over as indicated by an empty event queue.

Running a Thread of Control

To run a thread of control, we take one event off the thread of control event queue and process it. Then, we simply finish dispatching any events off of the imminent event queue.

However, because an error handler may have transferred control outside of the run time, we must keep track of the state data transaction. We call the state of the system where the domain populations are consistent to be the *black* state (analogous to the usage of "black box" in testing terminology). When the domain populations are in flux, during a thread of control, we call this the *white* state (again, analogous to the usage in testing terminology). We keep track of the state using a boolean value.

```
<<mrt static data>>=
static bool mrtInWhiteState ;
```

In the case of executing threads of control, if we find ourselves in the **white** state, we must first get back to the **black** state before starting a new thread of control.

```
<<mrt static functions>>=
static bool
mrtRunThreadOfControl(void)
{
    if (mrtInWhiteState) {                          // ❶
        mrtFinishThreadOfControl() ;
    }

    bool dispatched = mrtDispatchTOCEvent() ;
    if (dispatched) {
        mrtFinishThreadOfControl() ;
    }

    return dispatched ;
}
```

❶ *N.B.* we must check if previous transaction finished before starting a new one.

A thread of control is finished by dispatching all the events on the imminent event queue and then ending the transaction.

```
<<mrt forward references>>=
static void mrtFinishThreadOfControl(void) ;
```

```
<<mrt static functions>>=
static void
mrtFinishThreadOfControl(void)
{
    while (mrtDispatchEventFromQueue(&eventQueue)) {
        // N.B. empty loop body
    }
    mrtEndTransaction() ;
}
```

Dispatching a Thread of Control Event

Before each event which starts a thread of control is processed, all the synchronization functions must be invoked. This allows the foreground to inject new data and events into the system when it has a consistent population.

```
<<mrt forward references>>=
static bool mrtDispatchTOCEvent(void) ;
```

```
<<mrt static functions>>=
static bool
mrtDispatchTOCEvent(void)
{
    int status = sys_ctrl_dispatch_notifications() ;
    if (status < 0) {
        mrtFatalError(mrtPanic,
                "failed to dispatch background notifications, %d\n",
                status) ;
    }
```

```
    bool dispatched ;
    if (!mrtEventQueueEmpty(&tocEventQueue)) {
        mrtInWhiteState = true ;
        dispatched = mrtDispatchEventFromQueue(&tocEventQueue) ;
    } else {
        dispatched = false ;
    }
    return dispatched ;
}
```

The code for `mrtDispatchEventFromQueue` is shown later when we discuss event dispatch.

Managing Data

Before we discuss the details of how execution is sequenced, we will show how data is managed by the run-time. Execution sequencing is directed at class instances and it will be helpful to understand how instances are stored before we get to discussing how they are operated upon.

The run-time provides functions to support the basic lifetime of instances. We need to be able to:

- Create instances synchronously as part of an activity.
- Create instances asynchronously as part of a event dispatch.
- Delete instances synchronously as part of an activity.
- Delete instances asynchronously when the instance enters a final state.

Instance Data

For every class (and assigner), the code generator will declare a "C" structure whose members contain all the attributes and other elements of the class. The structure of each class is, in general, different. Each class can have different attributes and relationships and this is reflected in members of the "C" structure that is used for each class. From the point of view of the run-time and the operations provided by the run-time, instances can be treated the same. The view of a class instance by the run-time is shown below.

```
<<mrt internal aggregate types>>=
typedef struct mrtinstance {
    struct mrtclass const *classDesc ;
    MRT_AllocStatus alloc ;
    MRT_StateCode currentState ;
    MRT_RefCount refCount ;

#       ifndef MRT_NO_NAMES
    char const *name ;
#       endif /* MRT_NO_NAMES */
} MRT_Instance ;
```

`classDesc`
: A pointer to a class data structure. The class data structure contains information that is common to all instances of the class.

`alloc`
: A value that shows the allocation status of the memory for the instance. A value of 0, means the memory slot is not in use and can be allocated to a new instance. A negative value means the memory has been reserved but is not in active use as an instance. A positive value means the memory has been reserved and the instance is in active use.

currentState
: For those classes that have a state model, the `currentState` member holds a small integer number indicating the current state in which the instance resides. A value of `MRT_StateCode_IG` (-1) is used to show that the class does not have a state model.

refCount
: A counter value used to enforce referential integrity. At the end of a data transaction, this member is used as an accumulator of the number of times the instance is referenced in a relationship.

name
: An instance may have a name that it was given during the intial instance population. For instances not defined as part of the initial instance population, this member is set to `NULL`.

The `micca` code generator will facilitate this view of an arbitrary instance by inserting this structure as the first element of the "C" structure that is generated for each class. In this manner, a pointer to an arbitrary class instance can be cast to a pointer to a `MRT_Instance` with impunity.

Allocation Status Data Type

```
<<mrt internal simple types>>=
typedef int16_t MRT_AllocStatus ;
```

A class instance is nothing more than an element of a "C" array. The size of the array, and consequently the maximum number of instances of the class, is fixed at compile time. The `alloc` member of the instance structure is used to keep track of the status of the array elements as instances are created and deleted at run-time. We will also use this member to track *event-in-flight* errors. We will discuss this more below, but we need a way to insure that events that have been signaled are not delivered to instances that have been deleted.

Current State Data Type

```
<<mrt interface simple types>>=
typedef int8_t MRT_StateCode ;
```

The data type for the `currentState` member is just a small integer. By specifying 8 bits we limit the number of states of a state model to 127. That is an enormous number of states for a class state model. We use negative state numbers to indicate the non-transitioning actions that may occur when an event is dispatched.

```
<<mrt interface constants>>=
#define MRT_StateCode_IG    (-1)
<<mrt internal constants>>=
#define MRT_StateCode_CH    (-2)
```

We will also have need for a counter used in the enforcement of referential integrity.

Reference Count Data Type

```
<<mrt internal simple types>>=
typedef uint8_t MRT_RefCount ;
```

Class Data

All the behavior of data management and execution sequencing is completely determined by the values contained in the data structures supplied to the various functions of the run-time. This is distinct from some software architecture mechanisms that use generated code from a model compiler to implement some capabilities. Being completely data driven and separately compiled is a design goal of the run-time.

Since the behavior of all instances of a given class is the same, each class has a data structure that contains all the class invariant information.

First, we consider class attributes. We will have need to obtain attribute values in a generic way. From the platform model, we see that there are two types of attributes, independent and dependent. We will need to encode that difference.

Attribute Type Data Type

```
<<mrt internal simple types>>=
typedef enum {
    mrtIndependentAttr,
    mrtDependentAttr
} MRT_AttrType ;
```

Dependent attributes are those computed by some formula. The code generator will enclose the formula in a function that we can invoke to obtain the attribute value.

Attribute Formula Data Type

```
<<mrt internal aggregate types>>=
typedef void MRT_AttrFormula(
    void const *const self,
    void *const pvalue,
    MRT_AttrSize vsize) ;
```

self
A pointer to the instance upon which the formula function is invoked.

pvalue
A pointer to the memory where the formula result is to be placed.

vsize
The number of bytes pointed to by `pvalue`.

Formula functions will be invoked with a pointer to the instance, a pointer to where the result is to be stored and the size of the result area in bytes.

The description of an attribute is then a discriminated union.

```
<<mrt internal aggregate types>>=
typedef struct mrtattribute {
    MRT_AttrSize size ;
    MRT_AttrType type ;
    union {
        MRT_AttrOffset offset ;
        MRT_AttrFormula *formula ;
    } access ;

#       ifndef MRT_NO_NAMES
    char const *name ;
#       endif /* MRT_NO_NAMES */
} MRT_Attribute ;
```

size
The number of bytes of memory occupied by the attribute value.

type
The type of the attribute, independent or dependent.

access
A union whose value depends upon the value of the `type` member.

offset
For `mrtIndependentAttr` types, the `offset` member contains the offset in bytes from the beginning of the instance memory to the attribute.

formula
For `mrtDependentAttr` types, the `formula` member contains a pointer to a function that computes the value of the attribute.

name
The name of the attribute.

To manage all the aspects of class instances, we need information on memory allocation, event dispatch, relationships in which the instances participate and many other pieces of information.

```
<<mrt internal aggregate types>>=
typedef struct mrtclass {
    struct mrtinstallocblock *iab ;
    unsigned eventCount ;
    struct mrteventdispatchblock const *edb ;
    struct mrtpolydispatchblock const *pdb ;
    unsigned relCount ;
    struct mrtrelationship const * const *classRels ;
    unsigned attrCount ;
    MRT_Attribute const *classAttrs ;
    unsigned instCount ;
    struct mrtsuperclassrole const *containment ;

#       ifndef MRT_NO_NAMES
    char const *name ;
    char const *const *eventNames ;
#       endif /* MRT_NO_NAMES */
} MRT_Class ;
```

iab
A pointer to an instance allocation block (IAB). The IAB is used to dynamically allocation class instances.

eventCount
The total number of transitioning and polymorphic events to which the class responds.

edb
A pointer to an event dispatch block (EDB). The EDB is used to dispatch events to a state machine. For classes that do not have a state model, this value is set to `NULL`.

pdb
A pointer to a polymorphic dispatch block (PDB). The PDB is used to dispatch polymorphic events. For classes that have no polymorphic events, this value is set to `NULL`.

relCount
The number of relationships in which the class participates.

classRels
A pointer an array of relationship description pointers describing the relationships in which the class participates. The array has `relCount` elements.

attrCount
The number of attributes the class contains.

classAttrs
A pointer an array of descriptions for the attributes the class contains. The array has `attrCount` elements.

instCount
The number of instances of the class. This number represents the number of elements in the array that is used to store the class instances.

containment
For classes that are union subclasses, this member points to a descriptor for the immediate superclass in the generalization. For other classes, the value is `NULL`.

name
A pointer to a `NUL` terminated string containing the name of the class.

eventNames
: A pointer to an array of character pointers to the names of the class events. This information is used in tracing event dispatch.

When generically describing classes, the subclasses of union based generalizations pose a special situation. All other class instances are stored in an array. Union subclasses are stored in the structure of their related superclass. The essential information for a union subclass is the class of its ultimate superclass and the offset from the beginning of the superclass to where the instance is located. Note that a union subclass can be subject to repeated generalization of another union subclass. The ultimate superclass is the class at the top of the generalization hierarchy.

In the next section, the instance allocation block is described as we continue to define how data is managed by the run-time. Later, we describe the event dispatch block and polymorphic dispatch block when we discuss event dispatch.

Instance Allocation

There are three ways to create an instance:

1. Create an instance as part of an initial instance population.
2. Create an instance synchronously to the execution of some activity by invoking a function.
3. Create an instance asynchronously to the execution of some activity by sending an event.

Creating initial instances is handled during code generation using the population data specified when the domain is configured. The code generator arranges for initial instance values to be inserted as initializers of the class storage array. In this section, we are going to discuss synchronous instance creation. This is instance creation by a direct function invocation and when the function returns the instance is ready and available. Later we will discuss asynchronous instance creation which is instance creation by signaling an event.

Instance Allocation Block

The instances of a class are contained in a single array variable which serves as the memory pool for the instances. To support managing a pool of class instances, an **Instance Allocation Block**, or **IAB** for short, data structure is used to keep track of the memory pool. What we need is a data structure that describes the properties of the class instance memory pool.

```
<<mrt internal aggregate types>>=
typedef void (*MRT_InstCtor)(void *const) ;
typedef void (*MRT_InstDtor)(void *const) ;
```

```
<<mrt internal aggregate types>>=
typedef struct mrtinstallocblock {
    void *storageStart ;
    void *storageFinish ;
    void *storageLast ;
    MRT_AllocStatus alloc ;
    size_t instanceSize ;
    MRT_InstCtor construct ;
    MRT_InstDtor destruct ;
    unsigned linkCount ;
    MRT_AttrOffset const *linkOffsets ;
} MRT_iab ;
```

storageStart
: A pointer to the beginning of the memory where the instance storage pool is located. This is the array allocated to hold the instances of a class.

storageFinish
A pointer to one element beyond the end of the instance storage pool for the class. This pointer may not be dereferenced, of course, but provides the boundary marker for the end of the pool.

storageLast
A pointer to the instance that was last allocated. This is used as the starting point for allocating the next instance.

alloc
The next value of the allocation counter to be assigned to a newly allocated instance. This member is used to give a unique number (modulo the maximum value that can be held in the data type) to each allocation of the array element where an instance is stored. The number is used to diagnose run-time analysis errors.

instanceSize
The number of bytes of memory occupied by an instance.

construct
A pointer to a constructor function. If there is no constructor defined for the class, then the value of this member may be set to `NULL`.

destruct
A pointer to a destructor function. If there is no destructor defined for the class, then the value of this member may be set to `NULL`.

linkCount
The number of link pointer containers in the instance.

linkOffsets
A pointer to an array of offsets to the link pointer containers in an instance. The array contains `linkCount` elements.

Instance creation and deletion also supports a very simplified notion of construction and destruction for the instances. This is no where near as complicated or full featured as something in C++. Constructors and destructors take no arguments but are implicitly supplied a pointer to the instance when invoked.

Constructors and destructors are primarily useful for when the instance has a more complicated data structure as an attribute, as might be the case if the attribute data type is user defined. If you need to do complicated construction of instances, the preferred method is to do that with an instance based operation or as part of a state activity for an asynchronously created instance.

Finding Instance Memory

Before we can create an instance, we need to find memory for it.

```
<<mrt forward references>>=
static MRT_Instance *
mrtFindInstSlot(
    MRT_iab *iab) ;
```

iab
A pointer to the instance allocation block for the class for which instance memory is to be allocated.

`mrtFindInstSlot` searches for unused instance memory in the memory pool described by `iab`. It returns a pointer to the allocated memory if successful and `NULL` if no memory is available in the class pool.

The allocation algorithm is a simple sequential search starting at the last location that was allocated.

```
<<mrt static functions>>=
static MRT_Instance *
mrtFindInstSlot(
    MRT_iab *iab)
```

```
{
    assert(iab != NULL) ;
    assert(iab->storageLast < iab->storageFinish) ;
    /*
     * Search for an empty slot in the pool. Start at the next location after
     * where we last allocated an instance.
     */
    MRT_Instance *inst ;
    for (inst = mrtNextInstSlot(iab, iab->storageLast) ;
            inst->alloc != 0 && inst != iab->storageLast ;
            inst = mrtNextInstSlot(iab, inst)) {
        /* Empty Body */
    }
    /*
     * Check if we ended up on a slot that is free.
     */
    return inst->alloc == 0 ? inst : NULL ; // ❶

}
```

❶ If we wrap all the way around to where we started and still did not find an instance storage element whose `alloc` member was zero, then we have run out of space! That condition is indicated by returning `NULL`.

Finding the next element in the instance storage array involves performing the pointer arithmetic modulo the size of the array. Since the pool is allocated in a contiguous block of memory, we must wrap around the iterator when it passes the end of the storage pool. That is accomplished with the `mrtNextInstSlot()` function.

```
<<mrt forward references>>=
static inline void *mrtNextInstSlot(MRT_iab *iab, void *ptr) ;
```

The `mrtNextInstSlot` function returns the memory address of the next instance after the one pointed to by `ptr` in the class instance pool described by `iab`.

```
<<mrt static functions>>=
static inline void *
mrtNextInstSlot(
    MRT_iab *iab,
    void *ptr)
{
    ptr = (void *)((uintptr_t)ptr + iab->instanceSize) ; // ❶
    if (ptr >= iab->storageFinish) { // ❷
        ptr = iab->storageStart ;
    }
    return ptr ;
}
```

❶ Since the size of instance varies from class to class, we must take over scaling the pointer arithmetic by the size of the instance.

❷ Perform the wrap around if we cross over the boundary of the storage array.

Instance Containment

For most classes, the Instance Allocation Block describes everything needed about how instances are stored. However, classes that are subclasses of a union generalization do not have their own storage pool. Union subclass instances are stored as part of the instance structure of the containing superclass. The containment may be several levels deep as a union subclass can be the

superclass for another union generalization. The `MRT_Class` data structure contains the necessary information to deal with this. The following function forms the basis for treating the union subclasses the same as other classes.

```
<<mrt forward references>>=
static MRT_iab *
mrtGetStorageProperties(
    MRT_Class const *const classDesc,
    size_t *offsetptr) ;
```

classDesc
A pointer to the class data for which storage properties are computed.

offsetptr
A pointer to a location where the instance offset is returned. If not `NULL`, then a value is returned via the pointer that is the offset in bytes from the beginning of an instance to where the class storage begins.

The `mrtGetStorageProperties` function returns a pointer to the IAB that describes the storage containment for the class. For non-union subclasses, this is simply the IAB that describes the class storage pool. For union subclasses, the returned IAB will be for the ultimate superclass that contains the class instances.

```
<<mrt static functions>>=
static MRT_iab *
mrtGetStorageProperties(
    MRT_Class const *const classDesc,
    MRT_AttrOffset *offsetptr)
{
    assert(classDesc != NULL) ;
    MRT_iab *iab = classDesc->iab ;
    MRT_AttrOffset instanceOffset = 0 ; // ❶
    for (struct mrtsuperclassrole const *container = classDesc->containment ;
            container != NULL ;
            container = container->classDesc->containment) {// ❷
        instanceOffset += container->storageOffset ;
        iab = container->classDesc->iab ;
    }
    if (offsetptr) {// ❸
        *offsetptr = instanceOffset ;
    }

    return iab ;
}
```

❶ For non-union subclasses, the offset to the beginning of the instance data is always zero. Only union subclasses are stored with their superclass and will have a non-zero offset.

❷ We iterate up the generalization hierarchy to find the ultimate superclass that contains the class instances. At each step of the iteration, we accumulate the relative offset of each contained instance.

❸ The offset is returned only if requested.

We will also find occasion to want to compute the index into the storage pool for a particular instance. Storage pool indices make convenient identifiers of a class instance outside of a domain. Again we must allow that the instance may be a union subclass instance.

```
<<mrt internal external interfaces>>=
extern MRT_InstId
mrt_InstanceIndex(
    void const *instance) ;
```

instance
A pointer to a class instance.

The `mrt_InstanceIndex` function returns the array index value of the instance in its storage pool.

```
<<mrt external functions>>=
MRT_InstId
mrt_InstanceIndex(
    void const *instance)
{
    MRT_Instance const *instref = instance ;
    assert(instref != NULL) ;
    assert(instref->classDesc != NULL) ;

    MRT_AttrOffset offset ;
    MRT_iab *iab = mrtGetStorageProperties(instref->classDesc, &offset) ;
    assert(instance >= iab->storageStart && instance < iab->storageFinish) ;
    MRT_InstId index = (((uintptr_t)instance - offset) -
            (uintptr_t)iab->storageStart) / iab->instanceSize ;     // ❶
    return index ;
}
```

❶ The `instanceSize` member of the IAB is the number of bytes occupied by an instance. For union subclasses, the IAB returned by `mrtGetStorageProperties` will be for the enclosing supertype. So we have to make sure and subtract off the offset to the subclass before computing the index. For non-union subclass instances, `offset` here will be zero. Such are the complexities when taking over the pointer arithmetic and it is necessary to be generic.

The inverse of finding the instance index is to convert such an index into a reference to an instance.

```
<<mrt internal external interfaces>>=
extern void *
mrt_InstanceReference(
    MRT_Class const *const classDesc,
    MRT_InstId index) ;
```

classDesc
A pointer to the class description for the reference to be computed.

index
An array index into the class storage pool. If `index` out of bounds, then a fatal system error occurs.

The `mrt_InstanceReference` function returns a pointer to the instance in the class storage pool for the class described by `classDesc` and indexed by the value of `index`.

```
<<mrt external functions>>=
void *
mrt_InstanceReference(
    MRT_Class const *const classDesc,
    MRT_InstId index)
{
    void *instance = mrtIndexToInstance(classDesc, index) ;     // ❶
```

```
    if (instance == NULL) {
        mrtUnallocSlotError(index, classDesc) ;
    }

    return instance ;
}
```

❶ First, compute the instance from the index. Then we check if the instance is in use. For external callers, computing a reference that is not in use is a fatal error.

The common code used internally to compute an instance reference from an index doesn't test to see if the instance is in use. Internal callers take on that responsibility and most frequently it is not necessarily an error.

```
<<mrt static functions>>=
static void *
mrtIndexToInstance(
    MRT_Class const *const classDesc,
    MRT_InstId index)
{
    assert(classDesc != NULL) ;
    MRT_AttrOffset offset ;
    MRT_iab *iab = mrtGetStorageProperties(classDesc, &offset) ;
    void *instance = (void *)((uintptr_t)iab->storageStart +
            (index * iab->instanceSize) + offset) ;
    if (instance >= iab->storageFinish) {                    // ❶
        mrtNoInstSlotError(classDesc) ;
    }

    return (((MRT_Instance *)instance)->alloc <= 0) ? NULL : instance ;
}
```

❶ It is a fatal error to index outside the bounds of the class storage pool.

Creating an Instance

Creating an instance involves allocating memory to hold the instance. Normally, each class has its own pool of memory for its instance and the IAB and functions shown previously are used to manage that memory. However, instances of union-based subclasses have their memory allocated internal to the superclass instance. This means finding memory for a union subclass instance is different than finding memory for other classes. For the union subclass case, we must locate the correct place inside the instance memory for its related superclass instance. This difference is dealt with by using a separate function to create union subclass instances. Note there are two functions for each of the various types of creation operations.

We first start with a description of synchronously creating a normal, non-union-based instance.

```
<<mrt internal external interfaces>>=
extern void *
mrt_CreateInstance(
    MRT_Class const *const classDesc,
    MRT_StateCode initialState) ;
```

classDesc
A pointer to the class data for the instance to be created.

initialState
The state number into which the instance will be placed. For classes that do not have an associated state model this argument is ignored. For classes that do have an associated state model, the class will be created in the state given by `initialState`. If this argument is `MRT_StateCode_IG` or if the value given for `initialState` is not a valid state number for the class, then the instance is created in its default initial state.

`mrt_CreateInstance` allocates memory for an instance of the class described by `classDesc` and places the instance in the state given by `initialState`. No state activity is run as part of synchronous instance creation.

```
<<mrt external functions>>=
void *
mrt_CreateInstance(
    MRT_Class const *const classDesc,
    MRT_StateCode initialState)
{
    assert(classDesc != NULL) ;

    /*
     * Search for an empty slot in the pool.
     */
    MRT_iab *iab = classDesc->iab ;
    MRT_Instance *inst = mrtFindInstSlot(iab) ;
    if (inst == NULL) {
        mrtNoInstSlotError(classDesc) ;
    }
    /*
     * Record where we left off for the next allocation attempt.
     */
    iab->storageLast = inst ;
    /*
     * Initialize the memory for the instance.
     */
    mrtInitializeInstance(inst, classDesc, initialState) ;

    return inst ;
}
```

Union based subclasses pose special cases since the memory used to hold an instance is *not* allocated from a pool but rather is embedded in the instance of a superclass. A different function is required and additional parameters are used to determine the memory location of the subclass instance.

```
<<mrt internal external interfaces>>=
extern void *
mrt_CreateUnionInstance(
    MRT_Class const *const classDesc,
    MRT_StateCode initialState,
    MRT_Relationship const *const genRel,
    void *super) ;
```

classDesc
A pointer to the class data for the instance to be created. The class described by `classDesc` must be a subclass of the relationship described by `genRel`.

initialState
The state number into which the instance will be placed. For classes that do not have an associated state model this argument is ignored. For classes that do have an associated state model, the class will be created in the state given by `initialState`. If this argument is `MRT_StateCode_IG` or if the value given for `initialState` is not a valid state number for the class, then the instance is created in its default initial state.

genRel
A pointer to a relationship description for the generalization in which the instance is a union-based subclass. The `relType` field of `genRel` must be `mrtUnionGeneralization`.

super
A pointer to the superclass instance where the subclass instance is to be created. `super` must be an instance of the superclass described by `genRel`.

`mrt_CreateUnionInstance` allocates memory for an instance of the union-based subclass described by `classDesc` and places the instance in the state given by `initialState`. No state activity is run as part of synchronous instance creation. The instance is created in place in the memory pointed to by `super`.

The two additional parameters required by `mrt_CreateUnionInstance` are need to provide the memory of the created instance (`super`) and to determine the offset into that memory where the instance storage is located (`genRel`).

```
<<mrt external functions>>=
void *
mrt_CreateUnionInstance(
    MRT_Class const *const classDesc,
    MRT_StateCode initialState,
    MRT_Relationship const *const genRel,
    void *super)
{
    assert(classDesc != NULL) ;
    assert(genRel != NULL) ;
    assert(super != NULL) ;
    assert(genRel->relType == mrtUnionGeneralization) ;

    if (genRel->relType != mrtUnionGeneralization) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }
    MRT_UnionGeneralization const *const gen =
            &genRel->relInfo.unionGeneralization ;
    /*
     * Verify that the new subclass is actually a subclass of the
     * generalization.
     */
    mrtFindUnionGenSubclassCode(classDesc, gen->subclasses,
            gen->subclassCount) ;
    /*
     * Verify the super class instance is truly of the super class.
     */
```

```
    MRT_Instance *superInst = super ;
    if (superInst->classDesc != gen->superclass.classDesc) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }
    /*
     * Compute the location of the union subclass instance
     * within the superclass instance.
     */
    void *inst = (void *)((uintptr_t)super + gen->superclass.storageOffset) ;
    /*
     * Initialize the memory for the instance.
     */
    mrtInitializeInstance(inst, classDesc, initialState) ;

    return inst ;
}
```

Initializing Instance Memory

Once we have identified the memory for an instance, we can make is ready to use. This involves zeroing out the memory and then initialzing the members of the `MRT_Instance` structure. Any link list references must be initialized to empty and the constructor is run if there is one.

```
<<mrt static functions>>=
static void
mrtInitializeInstance(
    MRT_Instance *inst,
    MRT_Class const *const classDesc,
    MRT_StateCode initialState)
{
    assert(inst != NULL) ;
    assert(classDesc != NULL) ;

    MRT_iab *iab = classDesc->iab ;
    assert(iab != NULL) ;
    /*
     * Start with a zeroed out memory space.
     */
    memset(inst, 0, iab->instanceSize) ;                              // ❶
    inst->classDesc = classDesc ;
    /*
     * Mark the slot as in use.
     */
    inst->alloc = mrtIncrAllocCounter(iab) ;
    if (classDesc->edb != NULL) {
        assert(initialState < classDesc->edb->stateCount) ;

        inst->currentState = (initialState == MRT_StateCode_IG ||
            initialState >= classDesc->edb->stateCount) ?
                classDesc->edb->initialState : initialState ;         // ❷
    } else {
        inst->currentState = MRT_StateCode_IG ;                       // ❸
    }

    MRT_AttrOffset const *offsets = iab->linkOffsets ;                // ❹
    for (unsigned count = iab->linkCount ; count != 0 ; count--) {
        MRT_LinkRef *link = (MRT_LinkRef *)((uintptr_t) inst + *offsets++) ;
        mrtLinkRefInit(link) ;
    }
    /*
```

```
     * Run the constructor if there is one.
     */
    if (iab->construct) {
        iab->construct(inst) ;
    }
    mrtMarkRelationship(classDesc->classRels, classDesc->relCount) ; // ❺
}
```

❶ Setting the memory of the instance to zero is very important. This insures that backward referencing pointers that are used in relationships are `NULL` and we depend upon that fact in the relationship linkage code.

❷ We use `MRT_StateCode_IG` as a special value to indicate that we want the instance created in its default initial state. We also protect against illegal values of the initial state and treat them the same.

❸ If a class does not have a state model, then we ignore the value passed in to the function and set the current state to ignored. This is a convenient value for such a situation.

❹ If a class contains any linked list terminus used for relationship navigation, they must be initialized to show that the linked list is empty.

❺ Creating an instance means it must be evaluated for referential integrity at the end of the data transaction. This is discussed in the next chapter.

One other important point here. There is a counter in the IAB that is incremented each time an instance is allocated and this value is used in the `alloc` member of the instance. This is another part of the strategy to detect an *event-in-flight* error. This is described further below. The effect of running this counter is that every instance gets a different `alloc` member value (modulo the size of the counter variable) The increment has one little catch. The counter is signed and the positive and negative values are used differently. Here, we make sure the value remains positive.

```
<<mrt forward references>>=
static inline MRT_AllocStatus mrtIncrAllocCounter(MRT_iab *iab) ;
```

```
<<mrt static functions>>=
static inline MRT_AllocStatus
mrtIncrAllocCounter(
    MRT_iab *iab)
{
    /*
     * Catch any overflow
     */
    iab->alloc = (iab->alloc == INT16_MAX ? 1 : iab->alloc + 1) ;
    return iab->alloc ;
}
```

Deleting an Instance

The function, `mrt_DeleteInstance`, is used to synchronously destroy an instance. Note that union subclass instances are treated the same as other class instances. Returning the memory for an instance is accomplished by setting the value of its `alloc` member to zero and this is the same in both cases, regardless of the fact that union subclass instance do not have their own memory pool.

```
<<mrt internal external interfaces>>=
extern void
mrt_DeleteInstance(
    void *instref) ;
```

instref
A pointer to an instance to be destroyed.

The `mrt_DeleteInstance` function deletes the instance given by `instref`.

Just as there was a distinction between synchronous and asynchronous instance creation, there is a similar distinction for destruction. Asynchronous destruction happens as a result of an instance entering an *final* state and that is discussed further below. Here we are dealing with synchronous destruction of an instance.

Deleting an instance also implies that when the class of the instance has references to other instance to link relationships, these linkages must also be deleted.

```
<<mrt external functions>>=
void
mrt_DeleteInstance(
    void *instref)
{
    MRT_Instance *inst = instref ;
    assert(inst != NULL) ;
    if (inst == NULL || inst->alloc <= 0) {          // ❶
        return ;
    }
    MRT_Class const *const classDesc = inst->classDesc ;
    assert(classDesc != NULL) ;
    MRT_iab *iab = classDesc->iab ;
    assert(iab != NULL) ;
    /*
     * Unlink the instance from its relationships.
     */
    mrtDeleteLinks(classDesc->classRels, classDesc->relCount, instref) ;
    /*
     * Run the destructor, if there is one.
     */
    if (iab->destruct) {
        iab->destruct(inst) ;
    }
    /*
     * Mark the slot as free.
     */
    inst->alloc = 0 ;
}
```

❶ Don't delete any instance that is unallocated or is awaiting the delivery of a creation event. In the later case, we would cause an event in flight error. Since `mrt_CreateInstanceAsync` returns a reference to the created instance, it is possible that something foolish like a request to immediately delete it could happen. Won't happen when using the embedded commands, but ...

Deleting an instance is a simple matter. First, unlink the instance from its relationships. If there is a destructor, it is run. The slot is free when its `alloc` member has a value of 0. But beware, for designs that have complicated relationships among the classes, instance deletion can be very complicated, requiring much care that the interdependencies among classes are properly preserved. That work is not done here! It is the responsibility of the analysis model to take any actions necessary to preserve data integrity when deleting an instance.

Iterating Over Class Instances

It is a common operation to iterate over the instances of a class. This is done both in the internals of the run-time as well as by domain activity code. For example, domain code has the need to search for instances meeting certain criteria such as the value of an attribute. For small instance populations, that search can be implemented by iterating over the instances of the class and making the comparison to the criteria. Consequently, the run-time provides a general means to iterate over the instances of a class.

The interface to instance iteration follows familiar patterns of Start, More, Next and Get. We have designed the interface in this way to avoid having to return `NULL` as a special value to indicate the end of the iteration. First, we need a data structure to hold the information we need for iteration.

```
<<mrt interface aggregate types>>=
struct mrtinstanceiterator ;
typedef struct mrtinstanceiterator MRT_InstIterator ;

<<mrt internal aggregate types>>=
struct mrtinstanceiterator {
    void *instance ;
    MRT_Class const *classDesc ;
    MRT_iab *iab ;
} ;
```

instance
A pointer to the memory of the next instance in the iteration.

classDesc
A pointer to the class descriptor across which the iteration occurs.

iab
A pointer to the Instance Allocation Block for the class across which the iteration is to be performed.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstIteratorStart(
    MRT_InstIterator *iter,
    MRT_Class const *const classDesc) ;
```

iter
A pointer to a class instance iterator that is used to record the state of the iteration.

classDesc
A pointer to the class description for the class across which the iteration will happen.

The `mrt_InstIteratorStart` function is called to initialize an instance iterator to iterate over instances of the class described by `classDesc`.

```
<<mrt external functions>>=
void
mrt_InstIteratorStart(
    MRT_InstIterator *iter,
    MRT_Class const *const classDesc)
{
    assert(iter != NULL) ;
    assert(classDesc != NULL) ;
    iter->classDesc = classDesc ;

    MRT_AttrOffset instanceOffset ;
```

```
    MRT_iab *iab = mrtGetStorageProperties(classDesc, &instanceOffset) ;
    assert(iab != NULL) ;

    iter->iab = iab ;
    iter->instance = (void *)((uintptr_t)iab->storageStart + instanceOffset) ;
    MRT_Instance *instref = iter->instance ;
    if (instref->alloc <= 0 || instref->classDesc != iter->classDesc) { // ❶
        mrt_InstIteratorNext(iter) ;
    }
    return ;
}
```

❶ The iterator is designed to check if the instance is actually in use. Here we check the first instance and if it is not being used advance the iterator onward.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstIteratorMore(
    MRT_InstIterator *iter) ;
```

iter
A pointer to a class instance iterator.

The `mrt_InstIteratorMore` function returns a boolean value indicating if there are addition class instances which have not been visited. It returns `true` to indicate that the iterator references a valid class instance and `false` otherwise.

```
<<mrt external functions>>=
bool
mrt_InstIteratorMore(
    MRT_InstIterator *iter)
{
    assert(iter != NULL) ;
    return iter->instance < iter->iab->storageFinish ;
}
```

```
<<mrt internal external interfaces>>=
extern void *
mrt_InstIteratorGet(
    MRT_InstIterator *iter) ;
```

iter
A pointer to a class instance iterator.

The `mrt_InstIteratorGet` function returns a pointer to the class instance that is currently being visited in the iteration. It is not valid to invoke `mrt_InstIteratorGet` after the `mrt_InstIteratorMore` function has returned `false`. This function is analogous to dereferencing a pointer.

```
<<mrt external functions>>=
void *
mrt_InstIteratorGet(
    MRT_InstIterator *iter)
{
    assert(iter != NULL) ;
    return iter->instance ;
}
```

```
<<mrt internal external interfaces>>=
extern void
mrt_InstIteratorNext(
    MRT_InstIterator *iter) ;
```

iter
A pointer to a class instance iterator.

The `mrt_InstIteratorNext` function advances the class instance iterator to the next valid class instance. To be a valid class instance, the instance memory slot must be allocated and in active use.

```
<<mrt external functions>>=
void
mrt_InstIteratorNext(
    MRT_InstIterator *iter)
{
    assert(iter != NULL) ;

    MRT_iab *iab = iter->iab ;
    assert(iab != NULL) ;

    while (iter->instance < iab->storageFinish) { // ❶
        iter->instance = (void *)((uintptr_t)iter->instance + iab->instanceSize) ;
        MRT_Instance *instref = iter->instance ;
        if (instref->alloc > 0 && instref->classDesc == iter->classDesc) {// ❷
            break ;
        }
    }
    return ;
}
```

❶ Advancing the instance pointer to past the storage pool for the class indicates that we have visited all the instances.

❷ It may seem strange to test that the class descriptor for the instance matches that of the class we are iterating across. However, we must handle the case of union based subtypes which can be reclassified. In that case, we are iterating across all potential subclass and only want to stop on the ones that match our original intent.

Instance Sets

Another common model level operation is to accumulate a set of class instances. Most action languages are based on the concept of *selecting* a set of class instances based on some criteria (*e.g.* matching some value of an attribute) and then iterating over that set to perform an operation on each instance. When the operation is simple, this construct can be translated into an interation over the class instances performing the criteria test and operation together. This is to say that just because the action language statement implies determining the contents of an instance set does not mean that it must necessarily be implemented that way. For simple situations involving searching for a criteria and then immediately performing some operation, it is not necessary to accumulate a set first and then iterate over the set.

However, there are times when obtaining a set is crucial and cannot be replaced by simple iteration over the class instances. For example, iterating across a many-to-many association may result in visiting the same destination instance multiple times. If we are applying a non-idempotent operation to the destination instance, the what we really need is the set of related instances so as not to have duplicates. For those cases the run-time provides an instance set concept.

The design of the instance set has the goal of making it reasonable to allocate the sets as automatic variables on the stack. One could design the instance set to be a list of instance reference pointers. Unfortunately, such sets would occupy a significant amount of space. We want to be able to allocate instance sets as automatic variables to avoid the complication of managing the lifetime of set variables. Since our target environment is usually quite limited, large automatic variables are to be avoided.

The chosen design uses a bit vector. Since class instances are contained in an array, the index of a class instance in the storage pool array can be used as an index into a bit vector. The bit vector design trades off more processing to build and access the set, but makes many other set operations much easier. For example, since instance sets do not contain duplicates, the ability to arithmetically `OR` in a bit means we do not have to make any explicit tests to avoid inserting duplicate instances in the set.

The following is the data structure for an instance set.

```
<<mrt internal simple types>>=
typedef uint32_t MRT_SetWord ;

<<mrt interface aggregate types>>=
struct mrtinstanceset ;
typedef struct mrtinstanceset MRT_InstSet ;

<<mrt internal aggregate types>>=
struct mrtinstanceset {
    MRT_Class const *classDesc ;                                    // ❶
    MRT_SetWord instvector[(MRT_INSTANCE_SET_SIZE + MRT_SETWORD_BITS - 1) /
            MRT_SETWORD_BITS] ;                                     // ❷
} ;

<<mrt internal constants>>=
#define MRT_SETWORD_BITS    (sizeof(MRT_SetWord) * 8)
```

❶ An instance set records instances of one particular class. In other words, instance sets are typed to the class of instances they hold.

❷ The bit vector is allocated in `MRT_SetWord` words and we round up to insure we have enough space for all the instances.

Like all the other memory allocations in the run-time, we have to fix the maximum number of instance references that can be held in the set. By default we set that number to 128.

```
<<mrt interface constants>>=
#ifndef MRT_INSTANCE_SET_SIZE
#   define MRT_INSTANCE_SET_SIZE 128
#endif /* MRT_INSTANCE_SET_SIZE */
#if __STDC_VERSION__ >= 201112L
static_assert(MRT_INSTANCE_SET_SIZE > 0, "Instance set size must be > 0") ;
#endif /* __STDC_VERSION__ >= 201112L */
```

A function is provided to properly initalize an instance set variable.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetInitialize(
    MRT_InstSet *set,
    MRT_Class const *const classDesc) ;
```

`set`
A pointer to an instance set.

`classDesc`
A pointer to a class descriptor. The instance set will contain only instances of this class.

The `mrt_InstSetInitialize` function initialized the instance set data structure pointed to by `set` to prepare it to accept instances of the class described by `classDesc`.

```
<<mrt external functions>>=
void
mrt_InstSetInitialize(
    MRT_InstSet *set,
    MRT_Class const *const classDesc)
{
    assert(classDesc != NULL) ;
    set->classDesc = classDesc ;
    memset(set->instvector, 0, sizeof(set->instvector)) ;
}
```

The primary operation on instance sets is to add an instance.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetAddInstance(
    MRT_InstSet *set,
    void *instance) ;
```

set
A pointer to an instance set.

instance
A pointer to a class instance to be inserted into the set.

The `mrt_InstSetAddInstance` function inserts `instance` into the instance set pointed to by `set`. Attempts to add an instance already in the set are silently ignored. Attempts to add an instance that is not of the same class for which the set was initialized is also silently ignored.

```
<<mrt external functions>>=
void
mrt_InstSetAddInstance(
    MRT_InstSet *set,
    void *instance)
{
    assert(instance != NULL) ;
    if (instance == NULL) {
        return ;
    }
    MRT_Instance *instref = instance ;

    assert(instref->classDesc == set->classDesc) ;
    if (instref->classDesc != set->classDesc) {          // ❶
        return ;
    }
    unsigned instid = mrt_InstanceIndex(instance) ;
    assert(instid < MRT_INSTANCE_SET_SIZE) ;
    if (instid >= MRT_INSTANCE_SET_SIZE) {
        mrtFatalError(mrtInstSetOverflow, instid) ;
    }
    set->instvector[instid / MRT_SETWORD_BITS] |=
            (1 << (instid % MRT_SETWORD_BITS)) ;         // ❷
}
```

❶ Make sure we do not add instances of the wrong class. Instance indices are only unique within a given class.

❷ This bit twiddling selects the correct bit in the correct word of the bit vector. The correct word is given by the quotient of the index and the number of bits in a set vector word. The correct bit offset within a word is given by the modulus.

We are using divide and modulus operations with the full expectation that the compiler will recognize, at some level of optimization, that `MRT_SETWORD_BITS` is a power of two and transform the divide and modulus into bitwise operations.

We also need a means to remove an instance from the set.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetRemoveInstance(
    MRT_InstSet *set,
    void *instance) ;
```

set
A pointer to an instance set.

instance
A pointer to a class instance to be inserted into the set.

The `mrt_InstSetRemoveInstance` function removes `instance` from the instance set pointed to by `set`. Attempts to remove an instance that is not in the set are silently ignored.

```
<<mrt external functions>>=
void
mrt_InstSetRemoveInstance(
    MRT_InstSet *set,
    void *instance)
{
    assert(instance != NULL) ;
    if (instance == NULL) {
        return ;
    }
    MRT_Instance *instref = instance ;

    assert(instref->classDesc == set->classDesc) ;
    if (instref->classDesc != set->classDesc) {
        return ;
    }
    unsigned instid = mrt_InstanceIndex(instance) ;
    assert(instid < MRT_INSTANCE_SET_SIZE) ;
    if (instid >= MRT_INSTANCE_SET_SIZE) {
        mrtFatalError(mrtInstSetOverflow, instid) ;
    }
    set->instvector[instid / MRT_SETWORD_BITS] &=
            ~(1 << (instid % MRT_SETWORD_BITS)) ; // ❶
}
```

❶ More bit twiddling. The bitwise AND of the one's complement of a mask clears in the result any bit that was set in the mask.

We also provide a test for membership in an instance set.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstSetMember(
    MRT_InstSet *set,
    void *instance) ;
```

set
A pointer to an instance set.

instance
A pointer to a class instance to be tested for set membership.

The `mrt_InstSetMember` function determines if `instance` is a member of the instance set pointed to by `set`. The instance must be of the same class as that associated with `set`. The function returns `true` is `instance` is in the set and `false` otherwise.

```
<<mrt external functions>>=
bool
mrt_InstSetMember(
    MRT_InstSet *set,
    void *instance)
{
    assert(instance != NULL) ;
    if (instance == NULL) {
        return false ;
    }
    MRT_Instance *instref = instance ;

    assert(instref->classDesc == set->classDesc) ;
    if (instref->classDesc != set->classDesc) {
        return false ;
    }
    unsigned instid = mrt_InstanceIndex(instance) ;
    assert(instid < MRT_INSTANCE_SET_SIZE) ;
    if (instid >= MRT_INSTANCE_SET_SIZE) {
        mrtFatalError(mrtInstSetOverflow, instid) ;
    }
    MRT_SetWord w = set->instvector[instid / MRT_SETWORD_BITS] ;
    MRT_SetWord mask = (1 << (instid % MRT_SETWORD_BITS)) ;

    return (w & mask) != 0 ;
}
```

We also supply set operations, starting with determining if the instance set is empty.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstSetEmpty(
    MRT_InstSet *set) ;
```

set
A pointer to an instance set.

The `mrt_InstSetEmpty` returns `true` if the instance set pointed to by `set` contains no elements and `false` otherwise.

```
<<mrt external functions>>=
bool
```

```
mrt_InstSetEmpty(
    MRT_InstSet *set)
{
    assert(set != NULL) ;
    MRT_SetWord *pvect = set->instvector ;
    while (pvect < set->instvector + COUNTOF(set->instvector)) {
        if (*pvect++ != 0) {
            return false ;
        }
    }

    return true ;
}
```

A function to determine the number of members of the instance set is provided

```
<<mrt internal external interfaces>>=
extern unsigned
mrt_InstSetCardinality(
    MRT_InstSet *set) ;
```

set
A pointer to an instance set.

The `mrt_InstSetCardinality` returns `true` if the instance set pointed to by `set` contains no elements and `false` otherwise.

```
<<mrt external functions>>=
unsigned
mrt_InstSetCardinality(
    MRT_InstSet *set)
{
    assert(set != NULL) ;
    unsigned card = 0 ;
    for (MRT_SetWord *pvect = set->instvector ;
            pvect < set->instvector + COUNTOF(set->instvector) ; pvect++) {
        MRT_SetWord w = *pvect ;
        MRT_SetWord mask = 1 ;
        for (unsigned bit = MRT_SETWORD_BITS ; w != 0 && bit != 0 ; bit--) {
            if ((w & mask) != 0) {
                card++ ;
                w &= ~mask ; // ❶
            }
            mask <<= 1 ;
        }
    }

    return card ;
}
```

❶ Once we count the bit, we clear it. This allows us to short circuit the loop if no other bits in the word are set.

A test for set equality is also supplied.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstSetEqual(
    MRT_InstSet *set1,
    MRT_InstSet *set2) ;
```

set1
A pointer to an instance set.

set2
A pointer to an instance set.

The `mrt_InstSetEqual` returns true if instance set `set1` is equal to instance set `set2` and false otherwise. If `set1` and `set2` refer to different instance set, then false is returned.

```
<<mrt external functions>>=
bool
mrt_InstSetEqual(
    MRT_InstSet *set1,
    MRT_InstSet *set2)
{
    assert(set1 != NULL) ;
    assert(set2 != NULL) ;
    assert(set1->classDesc != NULL) ;
    assert(set2->classDesc != NULL) ;

    if (set1->classDesc != set2->classDesc) {
        return false ;
    }

    MRT_SetWord *src1 = set1->instvector ;
    MRT_SetWord *src2 = set2->instvector ;
    while (src1 < set1->instvector + COUNTOF(set1->instvector)) {
        if (*src1++ != *src2++) {
            return false ;
        }
    }

    return true ;
}
```

Basic set operations are also provided. We start with the set union operation.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetUnion(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result) ;
```

set1
A pointer to an instance set.

set2
A pointer to an instance set.

result
A pointer to an instance set where the result is placed.

The `mrt_InstSetUnion` function computes the set union of `set1` and `set2` storing the result in the set pointed to by `result`. If `set1` and `set2` do not refer to sets of the same class, then the result set is the same set as `set1`.

```
<<mrt external functions>>=
void
mrt_InstSetUnion(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result)
{
    assert(set1 != NULL) ;
    assert(set1->classDesc != NULL) ;
    assert(set2 != NULL) ;
    assert(set2->classDesc != NULL) ;
    assert(result != NULL) ;
    assert(set1->classDesc == set2->classDesc) ;

    result->classDesc = set1->classDesc ;
    if (set1->classDesc == set2->classDesc) {
        MRT_SetWord *dst = result->instvector ;
        MRT_SetWord *src1 = set1->instvector ;
        MRT_SetWord *src2 = set2->instvector ;
        while (dst < result->instvector + COUNTOF(result->instvector)) {
            *dst++ = *src1++ | *src2++ ;
        }
    } else {
        memcpy(result->instvector, set1->instvector, sizeof(result->instvector)) ;
    }
}
```

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetIntersect(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result) ;
```

set1
A pointer to an instance set.

set2
A pointer to an instance set.

result
A pointer to an instance set where the result is placed.

The `mrt_InstSetIntersect` function computes the set intersection of `set1` and `set2` storing the result in the set pointed to by `result`. If `set1` and `set2` do not refer to sets of the same class, then the returned set is the empty set of the same class as `set1`.

```
<<mrt external functions>>=
void
mrt_InstSetIntersect(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result)
{
    assert(set1 != NULL) ;
    assert(set1->classDesc != NULL) ;
    assert(set2 != NULL) ;
    assert(set2->classDesc != NULL) ;
    assert(result != NULL) ;
    assert(set1->classDesc == set2->classDesc) ;

    result->classDesc = set1->classDesc ;
    if (set1->classDesc == set2->classDesc) {
        MRT_SetWord *dst = result->instvector ;
        MRT_SetWord *src1 = set1->instvector ;
        MRT_SetWord *src2 = set2->instvector ;
        while (dst < result->instvector + COUNTOF(result->instvector)) {
            *dst++ = *src1++ & *src2++ ;
        }
    } else {
        memset(result->instvector, 0, sizeof(result->instvector)) ;
    }
}
```

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetMinus(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result) ;
```

set1
A pointer to an instance set.

set2
A pointer to an instance set.

result
A pointer to an instance set where the result is placed.

The `mrt_InstSetMinus` function computes the set difference of `set1` minus `set2` (order is significant in this case) storing the result in the set pointed to by `result`. If `set1` and `set2` do not refer to sets of the same class, then the result set is the same as `set1`.

```
<<mrt external functions>>=
void
mrt_InstSetMinus(
    MRT_InstSet *set1,
    MRT_InstSet *set2,
    MRT_InstSet *result)
{
    assert(set1 != NULL) ;
    assert(set1->classDesc != NULL) ;
    assert(set2 != NULL) ;
    assert(set2->classDesc != NULL) ;
    assert(result != NULL) ;
    assert(set1->classDesc == set2->classDesc) ;

    result->classDesc = set1->classDesc ;
    if (set1->classDesc == set2->classDesc) {
        MRT_SetWord *dst = result->instvector ;
        MRT_SetWord *src1 = set1->instvector ;
        MRT_SetWord *src2 = set2->instvector ;
        while (dst < result->instvector + COUNTOF(result->instvector)) {
            *dst++ = *src1++ & ~*src2++ ;
        }
    } else {
        memcpy(result->instvector, set1->instvector, sizeof(result->instvector)) ;
    }
}
```

Iterating Over Instance Sets

Once we have computed an instance set, we need some way to iterate over the resulting set. Since we have chosen a bit vector representation of the set, the iteration code is slightly more complicated as we must keep track of which word and bit offset we are currently referring to. Iterating over instance sets has one additional complication. It is possible that a class instance referenced in an instance set has been deleted in the time between when the instance set was accumulated and the time when the iteration across the set occurs. So, iterating over instance set must also guard against an instance that, although it may be a member of the set, has been deleted between the time the set was constructed and the time that the iteration occurs.

First, we start with the iterator data structure. This holds the state information we need to determine our place in the set.

```
<<mrt interface aggregate types>>=
```

```
struct mrtinstsetiterator ;
typedef struct mrtinstsetiterator MRT_InstSetIterator ;

<<mrt internal aggregate types>>=
struct mrtinstsetiterator {
    MRT_InstSet *set ;
    void *instance ;
    MRT_SetWord *vectorloc ;
    unsigned bitoffset ;
} ;
```

set
A pointer to the instance set across which the iteration will occur.

instance
A pointer to the current instance of the iteration.

vectorloc
A pointer to the word within the instance set where the current instance is located.

bitoffset
The bit offset into the word pointed to by `vectorloc` that represents the current instance of the iteration.

The operations on the iteration follow our usual pattern of Begin, More, Next and Get.

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetIterBegin(
    MRT_InstSet *set,
    MRT_InstSetIterator *iter) ;
```

set
A pointer to an instance set across which the iteration will occur.

iter
A pointer to an instance set iterator which will hold the state of the iteration.

The `mrt_InstSetIterBegin` initializes the set iterator pointed to by `iter` in order to iterate across the instance set pointed to by `set`. This function must be called first to prepare an iteration across an instance set.

```
<<mrt external functions>>=
void
mrt_InstSetIterBegin(
    MRT_InstSet *set,
    MRT_InstSetIterator *iter)
{
    assert(set != NULL) ;
    assert(iter != NULL) ;

    iter->set = set ;
    iter->vectorloc = set->instvector ;
    iter->bitoffset = 0 ;
    if ((*iter->vectorloc & 1) == 0) { // ❶
        mrt_InstSetIterNext(iter) ;
    } else {
        iter->instance = mrtIndexToInstance(iter->set->classDesc, 0) ;
        if (iter->instance == NULL) { // ❷
            mrt_InstSetIterNext(iter) ;
```

```
        }
    }
}
```

1. Check if the first instance belongs to the set. If not, we want to advance the iterator until we have a valid set member.
2. Check if the set instance is still actively being used. If not, we want to advance the iterator to one that is.

```
<<mrt internal external interfaces>>=
extern bool
mrt_InstSetIterMore(
    MRT_InstSetIterator *iter) ;
```

iter
A pointer to an instance set iterator.

The `mrt_InstSetIterMore` function returns `true` if there are more instances to visit in the instance set referenced by `iter` and `false` otherwise.

```
<<mrt external functions>>=
bool
mrt_InstSetIterMore(
    MRT_InstSetIterator *iter)
{
    assert(iter != NULL) ;
    return iter->instance != NULL ;
}
```

```
<<mrt internal external interfaces>>=
extern void *
mrt_InstSetIterGet(
    MRT_InstSetIterator *iter) ;
```

iter
A pointer to an instance set iterator.

The `mrt_InstSetIterGet` function obtains the current instance from the instance set referenced by `iter`. It is not valid to invoke this function after `mrt_InstSetIterMore` returns `false`.

```
<<mrt external functions>>=
void *
mrt_InstSetIterGet(
    MRT_InstSetIterator *iter)
{
    assert(iter != NULL) ;
    assert(iter->instance != NULL) ;
    return iter->instance ;
}
```

```
<<mrt internal external interfaces>>=
extern void
mrt_InstSetIterNext(
    MRT_InstSetIterator *iter) ;
```

iter
A pointer to an instance set iterator.

The `mrt_InstSetIterNext` function advances the set iterator pointed to by `iter` to the next element of the set.

Advancing the iterator is a more complicated operation. We must account for the word boundaries in the bit vector as well as the number of bits in each of the vector words. Finally, we must also make sure the next element of the set is still a valid, allocated instance of the class.

```
<<mrt external functions>>=
void
mrt_InstSetIterNext(
    MRT_InstSetIterator *iter)
{
    assert(iter != NULL) ;

    iter->bitoffset++ ; //❶
    if (iter->bitoffset >= MRT_SETWORD_BITS) {
        iter->vectorloc++ ;
        iter->bitoffset = 0 ;
    }
    while (iter->vectorloc <
            iter->set->instvector + COUNTOF(iter->set->instvector)) {
        if (*iter->vectorloc != 0) { // ❷
            MRT_SetWord mask = 1 << iter->bitoffset ;
            for ( ; iter->bitoffset < MRT_SETWORD_BITS ; iter->bitoffset++) {
                if ((*iter->vectorloc & mask) != 0) { // ❸
                    unsigned instindex =
                        (iter->vectorloc - iter->set->instvector) *
                        MRT_SETWORD_BITS + iter->bitoffset ; // ❹
                    void *instance = mrtIndexToInstance(iter->set->classDesc,
                            instindex) ;
                    if (instance != NULL) { // ❺
                        iter->instance = instance ;
                        return ;
                    }
                }
                mask <<= 1 ;
            }
        }
        iter->vectorloc++ ; // ❻
        iter->bitoffset = 0 ;
    }

    iter->instance = NULL ;
}
```

❶ Advance the iterator by one bit in the bit vector, accounting for running off the end of the vector word.

❷ A simple test determines if we can skip the entire word of bits.

❸ Check if we have found a set bit in the bit vector. This indicates that the corresponding instance is a member of the set.

❹ Compute the index of the instance in bit vector. This is the same to the index of the instance in the storage array.

❺ Verify that the instance was not deleted between the time the set was build and the time the iteration across the set happens.

❻ Advance to the next word of the bit vector.

Managing Referential Integrity

When the target translation platform is based on a Relational Database Management System (RDMS) or some other data architecture that supports the relational model of data (*e.g.* `rosea` or TclRAL), referential integrity checking comes with the underlying data management facilities. Such data management facilities constrain the values of data and move much of the data integrity validation away from the runtime code. Such is the motivation to use these facilities since they replace application code with declarative specifications of validity.

When targeting static-typed languages such as "C", translation platforms frequently do not offer any help in insuring referential integrity of the data as the system runs. Sometimes this limitation is understandable. It takes more code and data space to deal with referential integrity. For applications that do little dynamic instance creation or relationship changes, the risk that some code path results in a violation of referential integrity is much less. Nonetheless, execution platforms that do not provide any referential integrity checks impose a significant burden on the programmer and result in systems that are potentially less reliable.

In `micca`, we will support detection of referential integrity violations. As we will see, there is considerable complexity in achieving this goal. To be clear, this is what `micca` supports.

- Integrity constraints implied by the associations and generalizations supplied in the configuration DSL will be checked at run-time.
- Violations of referential integrity will result in a fatal system error. There is no support for rolling back data values to a known good state or ignoring the error and proceeding anyway.
- Integrity constraints are checked at the end of transactions on the data model. The duration of the transaction is not under direct program control, but there is an ongoing transaction when a thread of control is executing. A thread of control is defined to be the duration of the dispatch of the set of events initiated by an event signaled from outside of a state activity or by the delivery of a delayed event. We discuss the thread of control concept in more detail below when we deal with managing execution.

Because of our design choice to use the address of a class instance as an identifier, to manage referential integrity we need:

- Functions to create, delete and update reference pointers that implement relationships.
- Data structures and values that encode the specifications of the relationships from the platform model.
- Run-time code that, using the relationship specifications, evaluates whether referential integrity was maintained.

In keeping with our goal that the run-time be completely data driven, the code generator will supply the necessary information about all the associations and generalizations in the domain. This will allow us to manipulate the pointer values used in the relationships and to check that the result does not violate any of the multiplicity and conditionality constraints specified for the relationship.

We will show how this is accomplished by:

1. Showing the data structures used to describe relationships.
2. Explaining how transactions work and how referential integrity is checked.
3. Giving code to relate and unrelate instances in a relationship that operates with the integrity check.

Describing Relationships

In this section we present the data structures that are supplied by the `micca` code generator to describe the properties of a relationship. You can think of these data structures as an implementation version of the various classes in the platform model that deal with class relationships. The code generator takes data from the platform model population that was created during domain configuration and generates initialized "C" variables of the types described below.

From the platform model, we know that there are four different types of relationships.

```
<<mrt internal aggregate types>>=
typedef enum {
    mrtSimpleAssoc,
    mrtClassAssoc,
    mrtRefGeneralization,
    mrtUnionGeneralization
} MRT_RelType ;
```

The combination of the conditionality and multiplicity of a relationship can be encoded in four values.

```
<<mrt internal aggregate types>>=
typedef enum {
    mrtAtMostOne = 0,
    mrtExactlyOne,
    mrtZeroOrMore,
    mrtOneOrMore
} MRT_Cardinality ;
```

There are also three different ways that pointer references are stored.

```
<<mrt internal aggregate types>>=
typedef enum {
    mrtSingular,
    mrtArray,
    mrtLinkedList
} MRT_RefStorageType ;
```

For singular references, a single pointer member is allocated in the class structure.

For array reference storage, the class structure member is a counted array of the type shown below:

```
<<mrt internal aggregate types>>=
typedef struct mrtarrayref {
    MRT_Instance * const *links ;
    unsigned count ;
} MRT_ArrayRef ;
```

For linked list reference storage, the referring class has a set of link pointers that serve as the terminus for the linked list. In addition, the referenced class has a set of link pointers that enable it to be linked into the list of instances headed by the referring class.

```
<<mrt internal aggregate types>>=
typedef struct mrtlinkref {
    struct mrtlinkref *next ;
    struct mrtlinkref *prev ;
} MRT_LinkRef ;
```

Association Participants

To characterize the role a participant class in an association plays, the following data is needed.

```
<<mrt internal aggregate types>>=
typedef struct mrtassociationrole {
    MRT_Class const *classDesc ;
    MRT_Cardinality cardinality ;
    MRT_RefStorageType storageType ;
    MRT_AttrOffset storageOffset ;
    MRT_AttrOffset linkOffset ;
} MRT_AssociationRole ;
```

classDesc
A pointer to the class data for the participant.

cardinality
The encoding of the conditionality and multiplicity of the relationship. This encoding is from point of view of how many references to the participant that are made by the other participant in the relationship.

storageType
An encoding of the structure of how the reference pointers are stored.

storageOffset
The offset in bytes from the beginning of an instance to where the reference pointers for the relationship are stored.

linkOffset
For participants whose value of `storageType` is `mrtLinkedList`, this member gives the number of bytes from the beginning of the target participant instance where the linked list pointers are stored. For other values of `storageType` the value is ignored.

Simple Associations

Simple associations are made up of a source and target. The forward direction of a simple association is from the source instance to the target instance.

```
<<mrt internal aggregate types>>=
typedef struct mrtsimpleassociation {
    MRT_AssociationRole source ;
    MRT_AssociationRole target ;
} MRT_SimpleAssociation ;
```

Class Based Associations

For class based associations, there is another role played by the associator class. Associator classes always have two singular references to the participant classes.

```
<<mrt internal aggregate types>>=
typedef struct mrtassociatorrole {
    MRT_Class const *classDesc ;
    MRT_AttrOffset forwardOffset ;
    MRT_AttrOffset backwardOffset ;
    bool multiple ;
} MRT_AssociatorRole ;
```

classDesc
A pointer to the class data for the participant.

forwardOffset
The offset in bytes from the begining of an instance of the associator class to the forward reference pointer. The forward reference pointer refers to a class instance that serves as the target of the association.

backwardOffset
The offset in bytes from the begining of an instance of the associator class to the backward reference pointer. The backward reference pointer refers to a class instance that serves as the source of the association.

multiple
A boolean value that specifies if the same instances are allowed to be referenced by multiple association classes.

A class based association has descriptive information on all three class roles.

```
<<mrt internal aggregate types>>=
typedef struct mrtclassassociation {
    MRT_AssociationRole source ;
    MRT_AssociationRole target ;
    MRT_AssociatorRole associator ;
} MRT_ClassAssociation ;
```

Superclasses in a Generalization

The information needed to describe a superclass in a generalization is the same for both reference and union based generalizations.

```
<<mrt internal aggregate types>>=
typedef struct mrtsuperclassrole {
    MRT_Class const *classDesc ;
    MRT_AttrOffset storageOffset ;
} MRT_SuperClassRole ;
```

classDesc
A pointer to the class data for the participant.

storageOffset
The offset in bytes from the beginning of a superclass instance to where subclass information is stored. The information stored at `storageOffset` with either be a reference to the subclass instance (for reference based generalizations) or the union of the subclass structures in the generalization (for union based generalizations).

Reference Generalizations

For generalization relationships implemented by pointer references, the superclass implements its reference to a subclass by holding a simple pointer to the subclass instance. It is possible for the superclass to determine the type of the related subclass by simply following the reference to the subclass and then examining the `classDesc` member of the instance. data structure.

The information needed to describe the role of the subclass in a reference generalization is:

```
<<mrt internal aggregate types>>=
typedef struct mrtrefsubclassrole {
    MRT_Class const *classDesc ;
    MRT_AttrOffset storageOffset ;
} MRT_RefSubClassRole ;
```

classDesc
A pointer to the class data for the participant.

storageOffset
The offset in bytes from the beginning of a reference subclass instance to where the reference to its related superclass is stored. The structure member found at `storageOffset` from the beginning of the instance is a pointer to the superclass instance.

A reference generalization relationship can be described by its superclass role information and the set of subclass roles for the subclasses that participate in the generalization. We store the subclass role set as a count and pointer to a corresponding array.

```
<<mrt internal aggregate types>>=
typedef struct mrtrefgeneralization {
    MRT_SuperClassRole superclass ;
    unsigned subclassCount ;
    MRT_RefSubClassRole const *subclasses ;
} MRT_RefGeneralization ;
```

Union Generalizations

A union generalization results in subclass instances being held as a union in the superclass instance structure. From the superclass perspective, the information needed to describe where the subclass is stored is the same as for a reference supertype. The only difference is that `storageOffset` is an offset to the union holding the subclass rather than a reference pointer to a subclass.

Subclass instances held in union generalizations do not have a pointer back to their related superclasses. The related superclass can be determined by pointer arithmetic (*i.e.* subtracting the superclass `storageOffset` from the subclass instance pointer). So the role information for a union subclass is nothing more than a pointer to its class descriptor

Similar to the reference generalization, the union generalization information consists of that for the superclass an a set of subclass role information values.

```
<<mrt internal aggregate types>>=
typedef struct mrtuniongeneralization {
    MRT_SuperClassRole superclass ;
    unsigned subclassCount ;
    MRT_Class const * const *subclasses ;
} MRT_UnionGeneralization ;
```

Relationship Properties

Finally, the relationship data is held as a discriminated union. There is a type field and a union of four structures for the specific information required to describe each type of relationship.

```
<<mrt internal aggregate types>>=
typedef struct mrtrelationship {
    MRT_RelType relType ;
    union {
        MRT_SimpleAssociation simpleAssociation ;
        MRT_ClassAssociation classAssociation ;
        MRT_RefGeneralization refGeneralization ;
        MRT_UnionGeneralization unionGeneralization ;
    } relInfo ;

#       ifndef MRT_NO_NAMES
    char const *name ;
#       endif /* MRT_NO_NAMES */
} MRT_Relationship ;
```

`relType`
: A value to indicate the type of the relationship.

`relInfo`
: A union of the various types of relationship information structures that describes the details of the relationship. The union must be interpreted according to the value of the `relType` member.

`name`
: The name of the relationship.

Data Transactions

To check referential integrity, it is necessary to introduce the concept of a transaction on the domain data model. The transaction concept is necessary because we must be able to defer the integrity check until such time as the domain activities can insure that the integrity constraints are met. For example, consider two classes in a one-to-one unconditional association. If an instance of one class is created we do not want to check referential integrity until the code has had an opportunity to create a corresponding instance of the associated class.

There are two situations where we must enforce referential integrity:

1. After completion of a *thread of control*.
2. After completion of a *domain operation*.

We will describe how threads of control work later. For now, we are concerned with data transactions as a means of deciding when data integrity is to be verified.

For execution sequencing by event dispatch, the data integrity rules require each state activity to leave the domain data in a consistent state or to signal one or more events that, when dispatched, will ultimately bring the domain data into a consistent state. The later phrase of this rule implies that there a time where the domain data is allowed to be inconsistent while execution continues to bring it into a consistent state. For us, this implies that a thread of control is allowed to complete before the domain data is evaluated for consistency.

A domain operation is a form of synchronous service supplied by a domain. Like state activities, synchronous services exhibit run to completion semantics. Domain operations have full access to the underlying domain data and may also signal events which eventually initiate a new thread of control. Thus, domain operations must leave the domain data in a consistent state. However, there is a complication associated with domain operations.

- While executing a thread of control, an activity may invoke a domain operation. At that time, the domain data may be inconsistent. Note that threads of control do not *nest*, *i.e.* there is only ever one thread of control executing at any time because only one thread of execution is supported.
- Domain operations are not so restricted. Considering external entity operations, it is not possible to predict the chain of invocations of domain operations.
- Finally, the `micca` run time does not have any concept of a domain. Surprising as that may seem, from the point of view of the `micca` run time, the world is big flat and governed only by the addresses of descriptive data.

The implications of these considerations imply that data transactions must be nestable. Domain operations must start a new nested transaction and end it when they are done. Threads of control need not bother since they are not nestable in the single threaded `micca` run time. The code generator for `micca` does handle the nesting requirement, so there is no additional information required for the translation. It is just a matter of putting all domain operations within a nested data transaction and the code generation accomplishes that.

Synchronous Service Support

Two additional external functions are supplied by the run time to support synchronous operations on the domain, e.g. domain operations.

1. `mrt_BeginSyncService` marks the beginning of the synchronous service.
2. `mrt_EndSyncService` marks the end of the synchronous service and is the indication to the run time code that the referential integrity of the domain data is to be checked.

Normally, translated models do not have to use these functions. `Micca` generates code to surround automatically each domain operation with a start / end pair. However, bridge operations may choose to use them, particularly in conjunction with the `micca` portal operations. Portal operations do not enforce referential integrity and so bridges which use the portal operations that can potentially affect the data integrity must use these functions to mark a data transaction boundary.

```
<<mrt external interfaces>>=
extern void mrt_BeginSyncService(void) ;
```

The `mrt_BeginSyncService` function informs the run time that synchronous operations are starting on the domain.

The implementation of the `mrt_BeginSyncService` just increments a variable that contains the current transaction level.

```
<<mrt external functions>>=
void
mrt_BeginSyncService(void)
{
    mrtIncrTransLevel() ;
    mrtInWhiteState = true ;
}
```

```
<<mrt external interfaces>>=
extern void mrt_EndSyncService(void) ;
```

The `mrt_EndSyncService` function informs the run time that synchronous operations are ended on the domain. The run time uses this indication to enforce integrity constraints on the domain data.

The implementation of `mrt_EndSyncService` invokes `mrtEndTransaction()` to verify the referential integrity of the domain data. The transaction level is decremented to indicate that the nested transaction is complete.

```
<<mrt external functions>>=
void
mrt_EndSyncService(void)
{
    mrtEndTransaction() ;
}
```

We keep track of the transaction level as a simple unsigned counter.

```
<<mrt implementation simple types>>=
typedef uint16_t MRT_LevelCount ;
```

```
<<mrt implementation constants>>=
#define MRT_MAX_TRANS_LEVEL_COUNT     UINT16_MAX
```

A file static variable holds the current transaction level.

```
<<mrt static data>>=
static MRT_LevelCount mrtTransLevel ;
```

```
<<mrt implementation static inlines>>=
static inline
void
mrtIncrTransLevel(void)
{
    if (mrtTransLevel >= MRT_MAX_TRANS_LEVEL_COUNT) {
        mrtFatalError(mrtTransOverflow) ;       // ❶
    }
    mrtTransLevel += 1 ;
}
```

❶ Overflowing the transaction level counter indicates a serious problem.

```
<<mrt implementation static inlines>>=
static inline
void
mrtDecrTransLevel(void)
{
    if (mrtTransLevel != 0) {
        mrtTransLevel -= 1 ;        // ❶
    }
}
```

❶ We allow for unbalanced decrements. This is important to the transaction level algorithm since threads of control don't adjust the transaction nesting level. Most of the time a TOC will run at level 0. At the end it will attempt to decrement the level. We must make sure the transaction level does not underflow.

Recording Transaction Details

Each time a relationship is manipulated, by:

a. creating instances that participate in a relationship

b. deleting instances that participate in a relationship

c. updating references between participating instances

d. reclassifying subclass instances in a generalization

the relationship pointer references must be verified. As described below, the run-time provides functions to perform the relationship pointer operations. When a transaction is ongoing, we save a reference to the relationship information along with the current transaction level for each relationship that is manipulated. The affected relationships are stored in the following structure.

```
<<mrt implementation aggregate types>>=
typedef struct mrtTransEntry {
    struct mrtTransEntry *next ;
    MRT_LevelCount level ;
    MRT_Relationship const *relationship ;
} MRT_TransEntry ;
```

The individual transaction entries are managed on a linked list. We'll follow our usual pattern and have a free list and an in-use list.

```
<<mrt static data>>=
static MRT_TransEntry *mrtFreeTransEntries ;
static MRT_TransEntry *mrtTransEntries ;
```

Storage for the transaction entries is allocated as an array. In this case, we need transaction entries for the maximum nesting depth of transactions.

```
<<mrt static data>>=
static MRT_TransEntry mrtTransStorage[MRT_TRANSACTION_SIZE] ;
```

```
<<mrt implementation constants>>=
#ifndef MRT_TRANSACTION_SIZE
#   define MRT_TRANSACTION_SIZE 16
#endif /* MRT_TRANSACTION_SIZE */
```

To manage the transaction entries, we need functions to initialize the various lists and preform the usual list manipulations.

```
<<mrt implementation static inlines>>=
static inline
void
mrtInsertTrans(
    MRT_TransEntry **list,
    MRT_TransEntry *entry)
{
    assert(list != NULL) ;
    assert(entry != NULL) ;

    entry->next = *list ;
    *list = entry ;
}
```

```
<<mrt implementation static inlines>>=
static inline
MRT_TransEntry *
mrtRemoveTrans(
    MRT_TransEntry **list)
{
    assert(list != NULL) ;

    MRT_TransEntry *entry = *list ;
    if (entry != NULL) {
        *list = entry->next ;
        entry->next = NULL ;                // ❶
    }

    return entry ;
}
```

❶ Not strictly necessary, but useful during debugging.

After checking the referential integrity of an entry, it is discarded by removing it from the set of check entries and queue the entry back to the free queue for later use.

```
<<mrt implementation static inlines>>=
static inline
void
mrtDiscardTrans(void)
{
    MRT_TransEntry *entry = mrtRemoveTrans(&mrtTransEntries) ;
    assert(entry != NULL) ;                                    // ❶
    if (entry != NULL) {
        mrtInsertTrans(&mrtFreeTransEntries, entry) ;
    }
}
```

❶ We shouldn't be *over* discarding, but will tolerate it in a release build.

```
<<mrt static functions>>=
static void
mrtTransactionsInit(void)
{
    mrtFreeTransEntries = NULL ;
    mrtTransEntries = NULL ;
    mrtTransLevel = 0 ;
```

```
    MRT_TransEntry const *end = mrtTransStorage + MRT_TRANSACTION_SIZE ;
    for (MRT_TransEntry *iter = mrtTransStorage ;
            iter < end ; iter++) {
        mrtInsertTrans(&mrtFreeTransEntries, iter) ;
    }
}
```

Adding a relationship to the set to be checked is done by obtaining a free entry and inserting it onto the list of transaction entries to check.

```
<<mrt static functions>>=
static void
mrtAddRelToCheck(
    MRT_Relationship const *rel)
{
    MRT_TransEntry *entry = mrtRemoveTrans(&mrtFreeTransEntries) ;
    if (entry == NULL) {
        mrtFatalError(mrtTransOverflow) ;
    }

    entry->level = mrtTransLevel ;
    entry->relationship = rel ;
    mrtInsertTrans(&mrtTransEntries, entry) ;
}
```

During a transaction, run-time functions that either modify the relationship pointer storage or change the number of active instances invoke `mrtMarkRelationship` to save away the relationships that will need to be checked at the end of the transaction. This prevents us from having to check every relationship at the end of every transaction.

The only complication in the marking algorithm is that we wish to maintain the marked relationship descriptions as a set with no duplicates. Modifying instances of relationships multiple times during the transaction does not mean we need to evaluate the referential integrity more than once. We wish to save that computation and consequently will search the marked relationships to eliminate any duplicate mark.

```
<<mrt static functions>>=
static bool
mrtFindRelEntry(
    MRT_Relationship const *const rel)
{
    for (MRT_TransEntry *iter = mrtTransEntries ;
            iter != NULL && iter->level >= mrtTransLevel ;  // ❶
            iter = iter->next) {
        if (iter->relationship == rel) {
            return true ;
        }
        // N.B. no "else"
    }

    return false ;
}
```

❶ The search can be stopped when we cross over a transaction boundary. It is possible to have the same relationship modified in two different transactions, but there is no harm. We will just end up doing an extraneous check. It is more important to discovery any integrity problems in the current ongoing transaction.

```
<<mrt forward references>>=
static void
mrtMarkRelationship(
    MRT_Relationship const *const *rel,
    unsigned relCount) ;
```

Marking a relationship to be checked means searching the current set of marked relationships and, if we have not already added it, then it is inserted into our set to check.

```
<<mrt static functions>>=
static void
mrtMarkRelationship(
    MRT_Relationship const * const *rel,
    unsigned relCount)
{
    for ( ; relCount != 0 ; relCount--, rel++) {
        bool found = mrtFindRelEntry(*rel) ;
        if (!found) {
            mrtAddRelToCheck(*rel) ;
        }
    }
}
```

At the end of a transaction, we successively examine entries in our list and determine if they pass referential integrity. We perform the examination on the head of the list. If that succeeds, we discard the entry and continue looking by examining the entry left at the head. We are also only going to examine entries that are part of the current transaction.

The first failing check stops the whole process with a fatal error. The entry containing the relationship in error is left on the list. This is done because it is possible for applications to supply fatal error handlers which `longjmp()` out of the event loop[3]. The integrity problem still exists and could be repaired. In any case, we want that relationship to be evaluated again if we reenter the event loop. It is also useful for debugging since the faulting relationship is always at the head of the list.

```
<<mrt forward references>>=
static void mrtEndTransaction(void) ;

<<mrt static functions>>=
static void
mrtEndTransaction(void)
{
    for (MRT_TransEntry *rentry = mrtTransEntries ;
            rentry != NULL && rentry->level >= mrtTransLevel ;  // ❶
            rentry = mrtTransEntries) {
        if (mrtCheckRelationship(rentry->relationship)) {
            mrtDiscardTrans() ;
        } else {
            mrtDecrTransLevel() ;                               // ❷
            mrtRefIntegrityError(rentry->relationship) ;
        }
    }
    mrtDecrTransLevel() ;                                       // ❸
    if (mrtTransLevel == 0) {
        mrtInWhiteState = false ;
    }
}
```

❶ Again, we stop processing the entries in the list when we cross the boundary of a transaction.

❷ Since the integrity failure is a fatal error, we want to decrement the transaction level to show this evaluation is over. Some bridges, for testing specifically, will want to fix the integrity problem and be able to proceed on. This implies that failed checks become part of the next nested transaction or are caught when a thread of control terminates.

❸ We finish successfully and can mark the end of the transaction by decrementing the level.

[3] Test programs benefit from this arrangement

Verifying Referential Integrity

At the end of a transaction on the data model, `mrtEndTransaction` iterates across the set of saved relationship descriptors and checks the relationship to insure the data model is in a consistent state. The design approach is to use the `refCount` member of each instance as a place to store how many times the instance is referred to by the other participant in the relationship. Knowing the cardinality of the relationship, we can evaluate the `refCount` values to determine if referential integrity has been maintained.

The overall steps are:

1. Zero out the `refCount` member of all instances of both participating classes.
2. Using the relationship description information, find the reference pointers in the class instance that refer to the other participating class instances and increment the `refCount` member in the other participating instance if its pointer is found. This is done for both participants in the relationship.
3. Evaluate the `refCount` value against what the cardinality of the relationship requires.

This algorithm must be generic and capable of operating on any class instance. This will mean that there will be a lot of pointer arithmetic using offsets to structure members and other type unsafe operations[4] happening in the code. Sometimes "C" as a language is accused of being little more than a high level assembler and it is those type unsafe facilities that we must employ here to obtain a single body of code that can operate on an arbitrary class instance.

```
<<mrt forward references>>=
static bool mrtCheckRelationship(MRT_Relationship const *rel) ;
```

The `mrtCheckRelationship` function returns a boolean value indicating if the relationship described by `rel` has correct referential integrity.

Since there are four types of relationships in terms of how the reference pointers are stored, the function considers each type separately.

```
<<mrt static functions>>=
static bool
mrtCheckRelationship(
    MRT_Relationship const *rel)
{
    bool result = false ;

    switch (rel->relType) {
    case mrtSimpleAssoc: {
        <<mrtCheckRelationship: simple associations>>
    }
        break ;

    case mrtClassAssoc: {
        <<mrtCheckRelationship: class based associations>>
    }
        break ;

    case mrtRefGeneralization: {
        <<mrtCheckRelationship: reference generalizations>>
    }
        break ;

    case mrtUnionGeneralization: {
        <<mrtCheckRelationship: union generalizations>>
    }
        break ;
```

[4] By *type unsafe* we mean that the operations cannot be checked by the compiler to insure the type system isn't violated. The code itself is still standard "C" with well defined behavior.

```
    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }

    return result ;
}
```

For simple associations, we zero out the `refCount` member. The references between the class instances are tracked and then the values of the `refCount` members are compared against what the relationship requires. We do this in two steps, first tracking from source to target and then tracking from target to source. The reason for dividing it this way is to avoid stomping on the `refCount` in the case of a reflexive association. Since there is only one `refCount` member, we have to zero it, count the references and check the counts completely for one side before doing the same for the other. In a reflexive association, each side is the same class and so the `refCount` member is actually the same memory location for both sides.

```
<<mrtCheckRelationship: simple associations>>=
MRT_SimpleAssociation const *assoc = &rel->relInfo.simpleAssociation ;

MRT_Class const *const targetClass = assoc->target.classDesc ;
mrtZeroRefCounts(targetClass) ;                                    // ❶
mrtCountAssocRefs(&assoc->source, targetClass) ;                   // ❷

result = mrtCheckRefCounts(targetClass, assoc->target.cardinality) ;
if (!result) {
    break ;
}

MRT_Class const *const sourceClass = assoc->source.classDesc ;  // ❸
mrtZeroRefCounts(sourceClass) ;
mrtCountAssocRefs(&assoc->target, sourceClass) ;
result = mrtCheckRefCounts(sourceClass, assoc->source.cardinality) ;
```

❶ Start with the class that is the target of the association navigation.

❷ Count in the target side for each source side that references it.

❸ Now flip it around and do the target to source traversal.

Class based associations are more complicated. Recall that class based associations are treated in a decomposed manner where each side is related to the associator class and the associator class is unconditionally and singularly related to each participant. So we will consider each side separately.

```
<<mrtCheckRelationship: class based associations>>=
MRT_ClassAssociation const *assoc = &rel->relInfo.classAssociation ;

result = mrtCheckAssociatorRefs(&assoc->associator) ; // ❶
if (!result) {
    break ;
}
/*
 * On the first side, we evaluate the references from the source class
 * to the associator class.
 */
MRT_Class const *const assocClass = assoc->associator.classDesc ;
MRT_Class const *const sourceClass = assoc->source.classDesc ;

mrtZeroRefCounts(sourceClass) ;
mrtCountSingularRefs(assocClass, assoc->associator.backwardOffset,
        sourceClass) ;
```

```
result = mrtCheckRefCounts(sourceClass, assoc->source.cardinality) ;
if (!result) {
    break ;
}

mrtZeroRefCounts(assocClass) ;
mrtCountClassAssocRefs(&assoc->source, assocClass) ;

result = mrtCheckRefCounts(assocClass, mrtExactlyOne) ;
if (!result) {
    break ;
}
/*
* If the first side is okay, then we can evaluate the references from
* the target class to the associator class.
*/
MRT_Class const *const targetClass = assoc->target.classDesc ;

mrtZeroRefCounts(targetClass) ;
mrtCountSingularRefs(assocClass, assoc->associator.forwardOffset,
        targetClass) ;

result = mrtCheckRefCounts(targetClass, assoc->target.cardinality) ;
if (!result) {
    break ;
}

mrtZeroRefCounts(assocClass) ;
mrtCountClassAssocRefs(&assoc->target, assocClass) ;
result = mrtCheckRefCounts(assocClass, mrtExactlyOne) ;
```

❶ We start by checking the singular references from the associator class to the two participants.

Since a generalization represents a disjoint union between between the superclass and the set of subclasses, each superclass instance must be referenced exactly once from among all the subclasses of the generalization. Conversely, each subclass must reference exactly one superclass instance. This means we will have to iterate over the subclasses as we zero out the `refCount` member and count the references.

```
<<mrtCheckRelationship: reference generalizations>>=
MRT_RefGeneralization const *gen = &rel->relInfo.refGeneralization ;

mrtZeroRefCounts(gen->superclass.classDesc) ;

MRT_RefSubClassRole const *subclass = gen->subclasses ;
for (unsigned subcount = gen->subclassCount ; subcount != 0 ;
        subcount--, subclass++) {
    mrtZeroRefCounts(subclass->classDesc) ;
}

mrtCountGenRefs(gen) ;

result = mrtCheckRefCounts(gen->superclass.classDesc, mrtExactlyOne) ;

subclass = gen->subclasses ;
for (unsigned subcount = gen->subclassCount ; result && subcount != 0 ;
        subcount--, subclass++) {
    result = mrtCheckRefCounts(subclass->classDesc, mrtExactlyOne) ;
}
```

When a generalization is stored in a union, there are no reference pointers. So we don't have to follow any pointers to get to the subclass. Rather, the subclass in part of the superclass storage and pointer arithmetic computes the address of the subclass instance. However, the process is much the same as for reference generalizations. What is different is the way the references are counted.

```
<<mrtCheckRelationship: union generalizations>>=
MRT_UnionGeneralization const *gen = &rel->relInfo.unionGeneralization ;

mrtZeroRefCounts(gen->superclass.classDesc) ;

MRT_Class const *const *subclass = gen->subclasses ;
for (unsigned subcount = gen->subclassCount ; subcount != 0 ;
        subcount--, subclass++) {
    mrtZeroRefCounts(*subclass) ;
}

mrtCountUnionRefs(gen) ;                                                    // ❶

result = mrtCheckRefCounts(gen->superclass.classDesc, mrtExactlyOne) ;

subclass = gen->subclasses ;
for (unsigned subcount = gen->subclassCount ; result && subcount != 0 ;
        subcount--, subclass++) {
    result = mrtCheckRefCounts(*subclass, mrtExactlyOne) ;
}
```

❶ We need a different counting strategy to account for the union storage.

====== Zero Reference Counts

The first of the generic operations that must be performed on class instances is to set the `refCount` member of the instance structure to zero.

```
<<mrt forward references>>=
static void mrtZeroRefCounts(MRT_Class const *const classDesc) ;
```

The algorithm is simple. We iterate over all the class instances and set the `refCount` member to zero.

```
<<mrt static functions>>=
static void
mrtZeroRefCounts(
    MRT_Class const *const classDesc)
{
    MRT_InstIterator iter ;
    for (mrt_InstIteratorStart(&iter, classDesc) ; mrt_InstIteratorMore(&iter) ;
            mrt_InstIteratorNext(&iter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&iter) ;
        instref->refCount = 0 ;
    }
}
```

====== Checking Reference Counts

The other operation that depends only upon accessing the `refCount` member of an instance is to evaluate the value of `refCount` against what the cardinality of the relationship. There are four possible values of the cardinality. One of the values, `mrtZeroOrMore`, implies that any value of `refCount` is satisfactory since `refCount` is defined as an unsigned quantity and must, necessarily, be greater than or equal to zero.

For the other three values of cardinality we define functions to perform the comparison.

```
<<mrt static functions>>=
static bool
mrtCompareAtMostOne(
    unsigned refCount)
{
    return refCount <= 1 ;
}

static bool
mrtCompareExactlyOne(
    unsigned refCount)
{
    return refCount == 1 ;
}

static bool
mrtCompareOneOrMore(
    unsigned refCount)
{
    return refCount >= 1 ;
}
```

The reference counts are checked by iterating over the instances and invoking the proper cardinality comparison function. The first failure means we can stop.

```
<<mrt forward references>>=
static bool
mrtCheckRefCounts(MRT_Class const *const classDesc, MRT_Cardinality cardinality) ;
```

```
<<mrt static functions>>=
static bool
mrtCheckRefCounts(
    MRT_Class const *const classDesc,
    MRT_Cardinality cardinality)
{
    if (cardinality == mrtZeroOrMore) { // ❶
        return true ;
    }

    static bool (*const compareFuncs[])(unsigned) = {
        [mrtAtMostOne] = mrtCompareAtMostOne,
        [mrtExactlyOne] = mrtCompareExactlyOne,
        [mrtZeroOrMore] = NULL, // ❷
        [mrtOneOrMore] = mrtCompareOneOrMore
    } ;

    assert(cardinality <= mrtOneOrMore) ;
    bool (*const compareCardinality)(unsigned) = compareFuncs[cardinality] ; // ❸

    MRT_InstIterator iter ;
    for (mrt_InstIteratorStart(&iter, classDesc) ; mrt_InstIteratorMore(&iter) ;
            mrt_InstIteratorNext(&iter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&iter) ;
        if (!compareCardinality(instref->refCount)) {
            return false ; // ❹
        }
    }

    return true ;
}
```

❶ Dispense with the always true case first. There is no reason to iterate through the instances when the result is always true.

❷ We could omit this and it would be set to zero like any other uninitialized static variable. We include it to make clear that that we have accounted for all the cases and have factored out the `mrtZeroOrMore` case above.

❸ Select the comparison function based on the encoded cardinality. A small table of pointers to the comparison functions is handy to do this.

❹ We need not go past the first failure.

The singular references made by associator classes are special in the sense that they should never be `NULL`. Unrelating class based associations will set those references to `NULL` and we need to make sure that either the associator instance was reused by relating to other instances or it was deleted.

```
<<mrt forward references>>=
static bool
mrtCheckAssociatorRefs(MRT_AssociatorRole const *associator) ;
```

```
<<mrt static functions>>=
static bool
mrtCheckAssociatorRefs(
    MRT_AssociatorRole const *associator)
{
    MRT_InstIterator iter ;
    for (mrt_InstIteratorStart(&iter, associator->classDesc) ;
            mrt_InstIteratorMore(&iter) ; mrt_InstIteratorNext(&iter)) {
        void *inst = mrt_InstIteratorGet(&iter) ;
        MRT_Instance *ref = *(MRT_Instance **)
                ((uintptr_t)inst + associator->forwardOffset) ;
        if (ref == NULL || ref->alloc <= 0) { // ❶
            return false ;
        }
        ref = *(MRT_Instance **)((uintptr_t)inst + associator->backwardOffset) ;
        if (ref == NULL || ref->alloc <= 0) {
            return false ;
        }
    }

    return true ;
}
```

❶ Reference pointers in an associator class must be non-NULL and must point to an allocated instance.

Counting References

Of the four types of relationships, three of them actually contain pointer values that refer to class instances. The differences in the manner in which the three pointer references are organized leads us have separate functions for each type.

We use some common code to count the references made by an instance.

```
<<mrt implementation static inlines>>=
static inline
uint8_t
mrtIncrRefCount(
    uint8_t count)
{
    return (count == UINT8_MAX) ? 2 : count + 1 ;       // ❶
}
```

❶ We want to avoid the possibility of overflow of the reference counter. It is only an unsigned 8-bit quantity. Should it overflow, we could misinterpret the count. If we are at its max value, then we set it back to 2. Why 2? Zero and one are significant in this case and we know the values has already passed two, so we resume our count from there. Any value greater than 1 will yield the same conclusion about the referential integrity.

First, we consider counting the references in a simple association.

```
<<mrt forward references>>=
static void
mrtCountAssocRefs(
    MRT_AssociationRole const *source,
    MRT_Class const *const targetClass) ;
```

The `mrtCountAssocRefs` function counts the number of times each active instance of `source` refers to active instances of `targetClass`.

Counting references implies understanding how the reference pointers are stored. There are three ways that reference pointers are stored: single pointer, counted arrays and linked lists. We use the knowledge of offsets in the instance to where the pointer values are stored to access target instances.

```
<<mrt static functions>>=
static void
mrtCountAssocRefs(
    MRT_AssociationRole const *source,
    MRT_Class const *const targetClass)
{
    switch (source->storageType) {
    case mrtSingular:
        mrtCountSingularRefs(source->classDesc, source->storageOffset,
            targetClass) ;
        break ;

    case mrtArray:
        mrtCountArrayRefs(source->classDesc, source->storageOffset,
            targetClass) ;
        break ;

    case mrtLinkedList:
        mrtCountLinkedListRefs(source->classDesc, source->storageOffset,
                targetClass, source->linkOffset) ;
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

We will now show how the references stored in the three types are counted. For singular references, the reference is a pointer to the related class instance.

```
<<mrt forward references>>=
static void
mrtCountSingularRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset offset,
    MRT_Class const *const targetClass) ;
```

`sourceClass`
A pointer to the class description block for the class instances that make a singular reference to target instances.

offset
The offset in bytes from the beginning of source instances to where the pointer to the target instance is located.

targetClass
A pointer to the class description for the class instances to which the source instances refer.

The `mrtCountSingularRefs` functions iterates across all the active `source` instances, accesses the pointer in the source instance and increments the `refCount` member in the referenced target instance.

```
<<mrt static functions>>=
static void
mrtCountSingularRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset offset,
    MRT_Class const *const targetClass)
{
    MRT_iab *targetiab = mrtGetStorageProperties(targetClass, NULL) ;

    MRT_InstIterator srciter ;
    for (mrt_InstIteratorStart(&srciter, sourceClass) ;
            mrt_InstIteratorMore(&srciter) ; mrt_InstIteratorNext(&srciter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&srciter) ;
        MRT_Instance *targetInst =
                *(MRT_Instance **)((uintptr_t)instref + offset) ;
        if ((void *)targetInst >= targetiab->storageStart &&
                (void *)targetInst < targetiab->storageFinish &&
                targetInst->alloc > 0) { // ❶
            targetInst->refCount = mrtIncrRefCount(targetInst->refCount) ;
        }
    }
}
```

❶ Validate that the reference pointer value actually points into target instance storage. This will eliminate any NULL values also. Note we also insist that the target instance be active.

For array references, the references are an array of pointers to the related class instance.

```
<<mrt forward references>>=
static void
mrtCountArrayRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset offset,
    MRT_Class const *const targetClass) ;
```

sourceClass
A pointer to the class description block for the class instances that make an array reference to target instances.

offset
The offset in bytes from the beginning of source instances to where the pointer array to the target instance is located.

targetClass
A pointer to the class description for the class instances to which the source instances refer.

The `mrtCountArrayRefs` functions iterates across all the active `source` instances, accesses the pointer in the source instance and increments the `refCount` member in the referenced target instance.

```
<<mrt static functions>>=
static void
mrtCountArrayRefs(
    MRT_Class const *const sourceClass,
```

```
    MRT_AttrOffset offset,
    MRT_Class const *const targetClass)
{
    MRT_iab *targetiab = mrtGetStorageProperties(targetClass, NULL) ;

    MRT_InstIterator srciter ;
    for (mrt_InstIteratorStart(&srciter, sourceClass) ;
            mrt_InstIteratorMore(&srciter) ; mrt_InstIteratorNext(&srciter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&srciter) ;
        MRT_ArrayRef *srcrefs =
                (MRT_ArrayRef *)((uintptr_t)instref + offset) ;
        MRT_Instance *const *iter = srcrefs->links ;
        for (unsigned count = srcrefs->count ; count != 0 ; count--, iter++) {
            MRT_Instance *targetInst = *iter ;
            if ((void *)targetInst >= targetiab->storageStart &&
                    (void *)targetInst < targetiab->storageFinish &&
                    targetInst->alloc > 0) {
                targetInst->refCount = mrtIncrRefCount(targetInst->refCount) ;
            }
        }
    }
}
```

The structure of the linked list pointers makes counting them a bit more complicated. We must know the offset within the source instance where the link list terminus is located. Since the link pointers in the target are embedded somewhere in the target instance structure, we need to know the offset into the target instance that the links point to in order recover a pointer to the beginning of the target instance..

```
<<mrt forward references>>=
static void
mrtCountLinkedListRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset refOffset,
    MRT_Class const *const targetClass,
    MRT_AttrOffset linkOffset) ;
```

`sourceClass`
: A pointer to the class description for the class instances that make a linked reference to target instances.

`refOffset`
: The offset in bytes from the beginning of source instances to where the linked list terminus is located.

`targetClass`
: A pointer to the class description for the class instances to which the source instances refer.

`linkOffset`
: The offset in bytes from the beginning of target instances to where the linked list pointers are located.

The `mrtCountLinkedListRefs` function iterates through the linked list contained in the source instances to reference the target instances. When a target instance is found, it's `refCount` member is incremented.

```
<<mrt static functions>>=
static void
mrtCountLinkedListRefs(
    MRT_Class const *const sourceClass,
    MRT_AttrOffset refOffset,
    MRT_Class const *const targetClass,
    MRT_AttrOffset linkOffset)
{
    MRT_iab *targetiab = mrtGetStorageProperties(targetClass, NULL) ;
    MRT_InstIterator srciter ;
```

```
    for (mrt_InstIteratorStart(&srciter, sourceClass) ;
            mrt_InstIteratorMore(&srciter) ; mrt_InstIteratorNext(&srciter)) {
        MRT_Instance *srcInst = mrt_InstIteratorGet(&srciter) ;
        MRT_LinkRef *ref = (MRT_LinkRef *)((uintptr_t)srcInst + refOffset) ;
        for (MRT_LinkRef *trgIter = mrtLinkRefBegin(ref) ;
                !(trgIter == NULL || trgIter == mrtLinkRefEnd(ref)) ;    // ❶
                trgIter = trgIter->next) {
            MRT_Instance *targetInst =
                (MRT_Instance *)((uintptr_t)trgIter - linkOffset) ;      // ❷
            if ((void *)targetInst >= targetiab->storageStart &&
                    (void *)targetInst < targetiab->storageFinish &&
                    targetInst->alloc > 0) {
                targetInst->refCount = mrtIncrRefCount(targetInst->refCount) ;
            }
        }
    }
}
```

❶ Guard against uninitialized link pointers as well as detect the end of the linked list.

❷ The link pointers that form the linked list of target instances are located `linkOffset` from the beginning of the target instance. So we need to do some pointer arithmetic to get the pointer to the beginning of the instance. We have to do this computation because it may be the case that the target is on several linked lists depending upon the relationships in which it participates.

Counting class based associations uses the basic counting primatives that we have already seen.

```
<<mrt forward references>>=
static void
mrtCountClassAssocRefs(
    MRT_AssociationRole const *participant,
    MRT_Class const *const assocClass) ;
```

To count class based associations, we have to count the references from each participant to the associator class. References from the participant to the associator class can be of any type. References from the associator class back to the participant are always singular.

```
<<mrt static functions>>=
static void
mrtCountClassAssocRefs(
    MRT_AssociationRole const *participant,
    MRT_Class const *const assocClass)
{
    switch (participant->storageType) {
    case mrtSingular:
        mrtCountSingularRefs(participant->classDesc, participant->storageOffset,
            assocClass) ;
        break ;

    case mrtArray:
        mrtCountArrayRefs(participant->classDesc, participant->storageOffset,
            assocClass) ;
        break ;

    case mrtLinkedList:
        mrtCountLinkedListRefs(participant->classDesc,
                participant->storageOffset, assocClass,
                participant->linkOffset) ;
        break ;
```

```
    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

In a reference type generalization, all of the references are always singular. However, we must iterate across all the subclasses of the generalization in order to count the references that the subclasses make back to the superclass.

```
<<mrt forward references>>=
static void
mrtCountGenRefs(
    MRT_RefGeneralization const *gen) ;
```

```
<<mrt static functions>>=
static void
mrtCountGenRefs(
    MRT_RefGeneralization const *gen)
{
    MRT_Class const *const superClass = gen->superclass.classDesc ;
    MRT_InstIterator iter ;
    for (mrt_InstIteratorStart(&iter, superClass) ;                    // ❶
            mrt_InstIteratorMore(&iter) ; mrt_InstIteratorNext(&iter)) {
        MRT_Instance *instref = mrt_InstIteratorGet(&iter) ;
        MRT_Instance *subInst = *(MRT_Instance **)
                ((uintptr_t)instref + gen->superclass.storageOffset) ;
        if (subInst != NULL) {
            MRT_Class const *const subClass = subInst->classDesc ;

            MRT_iab *subiab = mrtGetStorageProperties(subClass, NULL) ;
            if ((void *)subInst >= subiab->storageStart &&
                    (void *)subInst < subiab->storageFinish &&
                    subInst->alloc > 0) {
                subInst->refCount = mrtIncrRefCount(subInst->refCount) ;
            }
        }
    }

    MRT_RefSubClassRole const *subclass = gen->subclasses ;
    for (unsigned subcount = gen->subclassCount ; subcount != 0 ;    // ❷
            subcount--, subclass++) {
        mrtCountSingularRefs(subclass->classDesc, subclass->storageOffset,
                gen->superclass.classDesc) ;
    }
}
```

❶ Start by iterating over the instances of the superclass.

❷ The references from the subclass to the superclass is just an ordinary singular pointer reference. We already know how to count them.

For a union based generalization, all the references are singular. However, since the storage for the subclass instance is part of the superclass instance storage, we only have to iterate across the superclass instances to have access to both reference counts.

```
<<mrt forward references>>=
static void
mrtCountUnionRefs(
    MRT_UnionGeneralization const *gen) ;
```

```
<<mrt static functions>>=
static void
mrtCountUnionRefs(
    MRT_UnionGeneralization const *gen)
{
    MRT_Class const *const superClass = gen->superclass.classDesc ;
    MRT_InstIterator superIter ;
    for (mrt_InstIteratorStart(&superIter, superClass) ;
            mrt_InstIteratorMore(&superIter) ;
            mrt_InstIteratorNext(&superIter)) {
        MRT_Instance *superInst = mrt_InstIteratorGet(&superIter) ;
        MRT_Instance *subInst = (MRT_Instance *)
                ((uintptr_t)superInst + gen->superclass.storageOffset) ;

        if (subInst->alloc > 0) {
            subInst->refCount = mrtIncrRefCount(subInst->refCount) ;
        }
    }

    MRT_Class const *const *subclass = gen->subclasses ;
    for (unsigned subcount = gen->subclassCount ; subcount != 0 ;
            subcount--, subclass++) {
        MRT_InstIterator subIter ;
        for (mrt_InstIteratorStart(&subIter, *subclass) ;
                mrt_InstIteratorMore(&subIter) ;
                mrt_InstIteratorNext(&subIter)) {
            MRT_Instance *subInst = mrt_InstIteratorGet(&subIter) ;
            MRT_Instance *superInst = (MRT_Instance *)
                    ((uintptr_t)subInst - gen->superclass.storageOffset) ;

            if (superInst->alloc > 0) {
                superInst->refCount = mrtIncrRefCount(superInst->refCount) ;
            }
        }
    }
}
```

Operations on Relationship Instances

Since the relationship descriptions have sufficient information to manage referential integrity, they also can be used to do the pointer manipulations required when instances of the relationships are created or deleted. It is important for the run-time to provide functions to perform these operations since checking referential integrity depends upon correct reference pointer manipulation.

Because we are using pointer values to manage instance identity and references[5], we need to be clear what operations must be performed for classes that participate in relationships.

For class based associations:

- There is a one-to-one correspondence between instances of a class based association and instances of the associator class itself.
- Because of this correspondence, when an instance of an associator is created, the creation operation must be supplied with instance references to the participating instances. The run-time will make up the pointer references appropriately. The participating instances must not already be part of the relationship. This implies that the participating instances are newly created or the run-time will unlink them from the association before creating the new instance of the association.
- Conversely, deleting an instance of an associator class is sufficient to delete an instance of the class based association itself.

[5] as opposed to attribute values if we were doing this using relational concepts

- Deleting an instance of a class participating in a class based association **does not** cause the corresponding instance (or instances) of the associator to be deleted.
- An operation to swap a different instance of a participating class must be provided.

For simple associations:

- There is a one-to-one correspondence between instances of the simple association and instances of the class which has the *referring* role.
- Because of this correspondence, when an instance of the referring class is created, the creation operation must be supplied with an instance reference to an instance of the class which plays the *referenced* role in the association.
- Conversely, deleting an instance of a referring class, unlinks the references to the referenced class.
- Deleting an instance of a referenced class, *does not* modify the reference in the referring class.
- An operation to swap a different instance of a referenced class must be provided.

For referenced based generalizations:

- When creating an instance of a subclass, the creation operation must be supplied with an instance reference to the superclass. The implication is that object creation must be in superclass to subclass order.
- Deleting an instance of the subclass, unlinks the references between the subclass and superclass. Deleting an instance of the superclass *does not* modify the reference in the subclass and the corresponding subclass instance is not deleted.
- An operation to reclassify subclass is also provided. This is a short hand for a delete / create sequence of a related subclass instance.

For union based generalizations:

- Subclass instances may not be directly created. It is necessary to create the superclass and reclassify the subclass instance to the desired subclass type. This may need to be repeated down a hierarchy if multiple generalization are defined. Note this implies the creation order for union based generalization is also from superclass to subclass order.
- Superclass or subclass instances may be deleted. In this case, there are no pointer references to modify, but referential integrity will be evaluated at the end of the data transaction.
- An operation to reclassify subclass is also provided. This is a short hand for a delete / create sequence of a related subclass instance.

Creating Relationship Links

The first part of supplying the operations on relationships is to be able to create the linkage between instances that refer to each other. We split this into two different functions, one for simple linkage and the other for associative linkage. Simple linkage forms the pointer references between only two participants in a relationship. This occurs for:

a. simple associations that are only between two participants,

b. associative classes when linking to only one participant, *i.e.* during a swap operation where one participant instance is being exchanged for another and

c. reference based generalizations

Associative linkage must account for the role of an associator class when linking together both participants in a class based association. This occurs when an instance of an associator class is created.

```
<<mrt internal external interfaces>>=
extern void
mrt_CreateSimpleLinks(
    MRT_Relationship const *const rel,
    void *const source,
    void *const target,
    bool isForward) ;
```

rel
A pointer to a relationship description for the relationship across which the relate operation is to happen.

source
A pointer to a class instance that serves the source role in the relationship.

target
A pointer to a class instance that serves the target role in the relationship.

isForward
A boolean to disambiguate the reflexive case. If `true`, then the link from `source` to `target` is in the forward direction. Otherwise, the link is to be made from `target` to `source`. This argument is ignored for relationships that are not reflexive class based associations.

This function is used to establish pointer links when instances which have simple association are created as well as when simple or class based association have their related instances "swapped".

For simple associations, `source` is an instance of the referring class and `target` is an instance of the referenced class. The `isForward` argument is ignored. For class based associations, `source` is an instance of the associator class and `target` is an instance of one of the participants. If the asssociation is reflexive, then the `isForward` argument determines if `target` is a target participant (true) or as source participant (false) . For reference generalizations, `source` is an instance of a subclass and `target` is an instance of the superclass. Union based generalizations have no links.

Since we have four different types of relationships (in terms of their storage properties), we consider each type as a separate case.

```
<<mrt external functions>>=
void
mrt_CreateSimpleLinks(
    MRT_Relationship const *const rel,
    void *const source,
    void *const target,
    bool isForward)
{
    assert(rel != NULL) ;
    assert(source != NULL) ;
    assert(target != NULL) ;

    switch (rel->relType) {
    case mrtSimpleAssoc: {
        <<mrt_CreateSimpleLinks: link simple association>>
    }
        break ;

    case mrtClassAssoc: {
        <<mrt_CreateSimpleLinks: link class association>>
    }
        break ;

    case mrtRefGeneralization: {
```

```
        <<mrt_CreateSimpleLinks: link reference generalization>>
    }
        break ;

    case mrtUnionGeneralization:
        // There are no pointer linkages for a union generalization.
        // N.B. fall through

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }

    mrtMarkRelationship(&rel, 1) ;      // ❶
}
```

❶ Since we are create links, we must mark the relationship to be evaluated at the end of the data transaction.

For simple associations, we must establish two sets of pointer linkages. From source to target is the primary referring direction. This is always singular and unconditional and so represented by a single pointer value. From target to source is in the back link direction and may be either singular or a linked list.

```
<<mrt_CreateSimpleLinks: link simple association>>=
MRT_SimpleAssociation const *const assoc = &rel->relInfo.simpleAssociation ;

MRT_Instance *srcInst = source ;
MRT_Instance *targetInst = target ;
if (assoc->source.classDesc != srcInst->classDesc ||
        assoc->target.classDesc != targetInst->classDesc) {
    mrtFatalError(mrtRelationshipLinkage) ;                          // ❶
}

void *currentTarget =
        *(void **)((uintptr_t)source + assoc->source.storageOffset) ;

if (currentTarget != target) {                                       // ❷
    if (currentTarget != NULL) {
        mrtUnlinkBackref(&assoc->target, source, currentTarget) ;    // ❸
    }
    mrtLink(&assoc->source, source, target) ;                        // ❹
    mrtLink(&assoc->target, target, source) ;
}
```

❶ Make sure that the relationship descriptive information matches the classes of the instances we were handed.

❷ Just guard against the case where we are linking together what is already linked. Call it a big no op.

❸ We unlink any back references by the current target instance. This in effect causes the the current target to be half unrelated.

❹ Create a link for each leg of the relationship — from source to target and then from target to source.

Since the core of what is going on here is linking the two instances, let's look at what `mrtLink` does. We will use this function several more times later on.

`MrtLink` creates one leg of a relationship link from one instance ("from") to another ("to"). Like most of the access to relationship pointers we have already seen, `mrtLink` does some unsafe pointer arithmetic to find the location in the instance where references are stored. Then, depending upon the type of reference storage, updates the instance pointers.

```
<<mrt static functions>>=
static void
mrtLink(
    MRT_AssociationRole const *const fromRole,
    void *const fromInst,
    void *const toInst)
{
    void *linkStorage = (void *)((uintptr_t)fromInst + fromRole->storageOffset) ;
    switch (fromRole->storageType) {
    case mrtSingular: {
        void **toLink = linkStorage ;
        *toLink = toInst ;                                          // ❶
    }
        break ;

    case mrtArray:
        // can't link array types
        mrtFatalError(mrtStaticRelationship) ;
        break ;

    case mrtLinkedList: {
        MRT_LinkRef *toLinks =
                (MRT_LinkRef *)((uintptr_t)toInst + fromRole->linkOffset) ;
        if (toLinks->next != NULL && toLinks->prev != NULL) {
            mrtLinkRefRemove(toLinks) ;                             // ❷
        }

        MRT_LinkRef *fromList = linkStorage ;                       // ❸
        mrtLinkRefInsert(toLinks, fromList) ;
    }
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

❶ For a singular pointer, overwriting the pointer value creates the link.

❷ If the "to" instances is already on a linked list somewhere, then we unlink it before placing it on the new list. We can do this since we do not need the list head to unlink an item from a list.

❸ The terminus of the back link list is in the `fromInst` and is located at the `storageOffset` within the instance. The link pointers in the `toInst` are at the `linkOffset` within the instance. Note the offset into `toInst` is actually given in the association description of the "from" instance. This arrangement is convenient in other cases, despite the confusion of using a from description to access something in the "to" instance.

For class based associations, the `mrt_CreateSimpleLinks` function updates only one of the pointer references, from the associator class to either the source or target participant of the relationship. The code is more complex here because we need to deal with several circumstances:

- The `source` argument is always the associative class, but we must figure out whether the `target` is a source participant instance or a target participant instance.
- We have to handle the reflexive case where the two participant classes are the same and the `isForward` argument resolves the direction.

- We must make sure that we are not creating a duplicate instance of the association. Since instances of an associative relationship correspond one-to-one to instances of the associator class, we have to make sure that there are no other associator class instances which have the same source and target participant instances. This is logically the same as enforcing an identifying constraint on the associator class since the referential attributes of an associator class form an identifier.

```
<<mrt_CreateSimpleLinks: link class association>>=
MRT_ClassAssociation const *const cassoc = &rel->relInfo.classAssociation ;
MRT_AssociatorRole const *const arole = &cassoc->associator ;
MRT_AssociationRole const *const srole = &cassoc->source ;
MRT_AssociationRole const *const trole = &cassoc->target ;

void *const associator = source ; // Change the variable names to keep things clear
void *const dest = target ;
if (arole->classDesc != ((MRT_Instance *)associator)->classDesc) { // ❶
    mrtFatalError(mrtRelationshipLinkage) ;
}

MRT_AssociationRole const *destrole ;
MRT_AttrOffset assocOffset ;
void *srcInst ;
void *targetInst ;

if (srole->classDesc == trole->classDesc) {                          // ❷
    // reflexive case
    if (trole->classDesc != ((MRT_Instance *)dest)->classDesc) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }
    if (isForward) {                                                 // ❸
        destrole = trole ;
        assocOffset = arole->forwardOffset ;
        srcInst = *(void **)((uintptr_t)associator + arole->backwardOffset) ;
        targetInst = dest ;
    } else {
        destrole = srole ;
        assocOffset = arole->backwardOffset ;
        srcInst = dest ;
        targetInst = *(void **)((uintptr_t)associator + arole->forwardOffset) ;
    }
} else {                                                             // ❹
    // non-reflexive case
    if (srole->classDesc == ((MRT_Instance *)dest)->classDesc) {
        // backward
        destrole = srole ;
        assocOffset = arole->backwardOffset ;
        srcInst = dest ;
        targetInst = *(void **)((uintptr_t)associator + arole->forwardOffset) ;
    } else if (trole->classDesc == ((MRT_Instance *)dest)->classDesc) {
        // forward
        destrole = trole ;
        assocOffset = arole->forwardOffset ;
        srcInst = *(void **)((uintptr_t)associator + arole->backwardOffset) ;
        targetInst = dest ;
    } else {
        mrtFatalError(mrtRelationshipLinkage) ;
    }
}

void **p_assocRef = (void **)((uintptr_t)associator + assocOffset) ;
if (*p_assocRef != NULL) {
    mrtUnlinkBackref(destrole, associator, *p_assocRef) ;            // ❺
    *p_assocRef = NULL ;
}
```

```
if (srcInst != NULL && targetInst != NULL && arole->multiple == false) {
    mrtCheckDupAssociator(rel, srcInst, targetInst) ;                    // ❻
}

*p_assocRef = dest ;                                                      // ❼
mrtLink(destrole, dest, associator) ;
```

❶ When invoked on a class based association, we insist that the "source" instance reference argument be an instance reference to the associator class. We introduce a new variable to prevent confusion of the `source` argument with the fact that it is now assumed to be an instance of the associator class.

❷ Check for the reflexive case.

❸ For the reflexive case, the `isForward` argument determines the direction of the destination of the link.

❹ For the non-reflexive case we can determine the direction since the classes of the two ends are different.

❺ Unlink the references between the associator and the destination, if any. This is a necessary step before we check for a duplicated associator instance. If this is really just an update of an existing association, then be must delete that association before testing that creating a new one would be a duplicate. Otherwise, link values laying around in the association class data structure could be interpreted as being part of a duplicate instance.

❻ Check that we are not duplicating an instance of the association. It is a fatal error to attempt to do so. Note we check that the source and target actually have valid instance pointers. It is possible for the association class **not** to be linked to anything (e.g. when it is first created) and so the source and/or target instance pointers may be `NULL`.

❼ Link the leg from the associative class to one of the participant instances and then in the opposite direction — from the participant instance to the associative class instance.

When relating reference generalizations, the reference from the subclass to the superclass is singular and unconditional. The reference from the superclass to the subclass is also singular.

```
<<mrt_CreateSimpleLinks: link reference generalization>>=
MRT_RefGeneralization const *const gen = &rel->relInfo.refGeneralization ;

MRT_Class const *const subclassClass =
        ((MRT_Instance *)source)->classDesc ;                            // ❶
int subclassCode = mrtFindRefGenSubclassCode(subclassClass, gen->subclasses,
        gen->subclassCount) ;

MRT_Class const *const superclassClass =
        ((MRT_Instance *)target)->classDesc ;                            // ❷
if (gen->superclass.classDesc != superclassClass) {
    mrtFatalError(mrtRelationshipLinkage) ;
}

void **p_superRef = (void **)((uintptr_t)source +
        gen->subclasses[subclassCode].storageOffset) ;
*p_superRef = target ;

void **p_subRef = (void **)((uintptr_t)target + gen->superclass.storageOffset) ;
*p_subRef = source ;
```

❶ For reference generalizations, the "source" instance reference must be one of the subclasses that participates in the relationship. The primary reference in a generalization is from subclass instance to superclass instance.

❷ The "target" instance reference must then be to the superclass of the generalization.

Finding the participating subclass of the generalization is a sequential search of the subclass roles in the relationship description.

```
<<mrt static functions>>=
static int
mrtFindRefGenSubclassCode(
    MRT_Class const *const subclassClass,
    MRT_RefSubClassRole const *subclasses,
    unsigned count)
{
    int subcode ;

    for (subcode = 0 ; subcode < count ; subcode++, subclasses++) {
        if (subclassClass == subclasses->classDesc) {
            return subcode ;
        }
    }

    mrtFatalError(mrtRelationshipLinkage) ;
}
```

Create Associator Links

Linking class based associations must account for the nature of the associator class. A class based association is treated as being decomposed into two associations, one each between the participants and the associator class. The associator class has singular, unconditional references to each participant. The participants have back references to the associator whose type depends upon multiplicity and the static nature of the association.

```
<<mrt internal external interfaces>>=
extern void
mrt_CreateAssociatorLinks(
    MRT_Relationship const *rel,
    void *assoc,
    void *source,
    void *target) ;
```

rel
A pointer to a relationship description for the relationship across which the relate operation is to happen.

assoc
A pointer to a class instance that serves the associator role in the relationship.

source
A pointer to a class instance that serves the source role in the relationship.

target
A pointer to a class instance that serves the target role in the relationship.

```
<<mrt external functions>>=
void
mrt_CreateAssociatorLinks(
    MRT_Relationship const *rel,
    void *assoc,
    void *source,
    void *target)
{
    assert(rel != NULL) ;
    assert(assoc != NULL) ;
    assert(source != NULL) ;
    assert(target != NULL) ;
```

```
    assert(rel->relType == mrtClassAssoc) ;
    if (rel->relType != mrtClassAssoc) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }

    MRT_ClassAssociation const *cassoc = &rel->relInfo.classAssociation ;
    MRT_AssociatorRole const *arole = &cassoc->associator ;

    assert(arole->classDesc == ((MRT_Instance *)assoc)->classDesc) ;
    if (arole->classDesc != ((MRT_Instance *)assoc)->classDesc) {
        mrtFatalError(mrtRelationshipLinkage) ;
    }

    if (!arole->multiple) {
        mrtCheckDupAssociator(rel, source, target) ;    // ❶
    }

    mrt_CreateSimpleLinks(rel, assoc, source, false) ;  // ❷
    mrt_CreateSimpleLinks(rel, assoc, target, true) ;
}
```

❶ Problem arises when source and target are already related to each other through some other associator class instance. If that is the case, then we violate the identity contraint on the associator class and we would not have proper sets. So we have to navigate from source to see if we can find target.

❷ In keeping with the concept that a class based assocation is decomposed into two association legs, we need only create the links for both sides.

To make sure that there are no duplicated association instances, we must check that for any given pair of instances across a class based association, that we cannot traverse the association and find a match. The strategy used by `mrtCheckDupAssociator` is to start at the source instance and traverse the relationship in search of a matching target instance. This technique saves looking at all the associator class instances and searching for both the source and target pointer values. That code is easier to write, but will usually examine every instance of the associator class (since finding a duplicate is a rare happening). Each target instance linked to the source instance through the associator class must be compared to the target instance which is about to be set. If a match is found, then this is an attempt to create a duplicate instance of the association and a fatal error is declared. The complication arises in that we have to traverse the association using data found in the relationship description, *i.e.* using descriptive meta data. This means we have to perform all the pointer arithmetic ourselves.

The implementation considers the three cases that arise as a result of the three different ways that pointer linkage is stored. Of course, array storage used for static associations is not modified and so results in a fatal error.

```
<<mrt static functions>>=
static void
mrtCheckDupAssociator(
    MRT_Relationship const *rel,
    void *source,
    void *target)
{
    assert(rel != NULL) ;
    assert(rel->relType == mrtClassAssoc) ;
    assert(source != NULL) ;
    assert(target != NULL) ;

    MRT_ClassAssociation const *cassoc = &rel->relInfo.classAssociation ;
    MRT_AssociatorRole const *arole = &cassoc->associator ;
    MRT_AssociationRole const *srole = &cassoc->source ;

    switch (srole->storageType) {
    case mrtSingular: {
```

```
        <<mrtCheckDupAssociator: singular reference>>
    }
        break ;
    case mrtArray:
        mrtFatalError(mrtStaticRelationship) ;
        break ;

    case mrtLinkedList: {
        <<mrtCheckDupAssociator: linked list reference>>
    }
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

If the source instance has a singular back link to the associator class instance there is only one possible target to which it is related.

```
<<mrtCheckDupAssociator: singular reference>>=
void *assocInst = *(void **)((uintptr_t)source + srole->storageOffset) ;
if (assocInst != NULL) {
    void *currentTarget =
            *(void **)((uintptr_t)assocInst + arole->forwardOffset) ;
    if (currentTarget == target) {
        mrtDupAssociatorError(rel) ;
    }
}
```

If the source instance has a linked list of back references to the associator class instance, then we have to set up an interation over the list to track down the related targets.

```
<<mrtCheckDupAssociator: linked list reference>>=
MRT_LinkRef *linksList =
        (MRT_LinkRef *)((uintptr_t)source + srole->storageOffset) ;     // ❶
for (MRT_LinkRef *assocLink = mrtLinkRefBegin(linksList) ;
        !(assocLink == NULL || assocLink == mrtLinkRefEnd(linksList)) ;
        assocLink = assocLink->next) {
    assert(srole->linkOffset != 0) ;
    void *assocInst = (void *)((uintptr_t)assocLink - srole->linkOffset) ;// ❷
    void *currentTarget = *(void **)((uintptr_t)assocInst + arole->forwardOffset) ;
    if (currentTarget == target) {
        mrtDupAssociatorError(rel) ;
        break ;
    }
}
```

❶ The linked list terminus for the back links is in the source instance.

❷ Recover the pointer to the beginning of the associative instance by subtracting off the offset to where the linked list pointers are stored.

Deleting Relationship Linkage

We must also provide a function to delete relationship linkage pointers.

```
<<mrt forward references>>=
static void mrtDeleteLinks(MRT_Relationship const * const *classRels,
        unsigned relCount, void *inst) ;
```

classRels
A pointer to an array of relationships description pointers that describe the relationships in which `inst` is a participant.

relCount
The number of elements in the `classRels` array.

inst
A pointer to the class instance that is to be unlinked from its relationship.

The implementation for deleting relationship links follows a similar pattern as for creating them. There are four distinct types of relationships and so we must consider each one.

```
<<mrt static functions>>=
static void
mrtDeleteLinks(
    MRT_Relationship const * const *classRels,
    unsigned relCount,
    void *inst)
{
    assert(inst != NULL) ;
    /*
     * Mark the transaction since we are updating the reference pointers.
     */
    mrtMarkRelationship(classRels, relCount) ;

    MRT_Instance *instref = inst ;
    MRT_Class const *const instclass = instref->classDesc ;

    for ( ; relCount != 0 ; relCount--, classRels++) {
        struct mrtrelationship const * const rel = *classRels ;
        switch (rel->relType) {
        case mrtSimpleAssoc: {
            <<mrtDeleteLinks: unlink simple association>>
        }
            break ;

        case mrtClassAssoc: {
            <<mrtDeleteLinks: unlink class based association>>
        }
            break ;

        case mrtRefGeneralization: {
            <<mrtDeleteLinks: unlink reference generalization>>
        }
            break ;

        case mrtUnionGeneralization:
            // For a union generalization, there are no pointer links.
            break ;

        default:
            mrtFatalError(mrtRelationshipLinkage) ;
            break ;
        }
    }
```

```
}
```

To unrelate simple associations, we must verify that the classes of the instance actually participate in the association and the perform the pointer operations to unlink the instances.

```
<<mrtDeleteLinks: unlink simple association>>=
MRT_SimpleAssociation const *assoc = &rel->relInfo.simpleAssociation ;

if (instclass == assoc->source.classDesc) {
    <<mrtDeleteLinks: unlink simple forward>>
} else if (instclass == assoc->target.classDesc) {
    <<mrtDeleteLinks: unlink simple backward>>
} else {
    mrtFatalError(mrtRelationshipLinkage) ;
}
```

Unlinking a simple association in the forward direction implies that we are unlinking the primary reference. This primary reference is always singular and always non-NULL. When we are unlinking the primary reference we will follow the reference and try to unlink any back references.

```
<<mrtDeleteLinks: unlink simple forward>>=
MRT_AssociationRole const *sourceRole = &assoc->source ;

if (sourceRole->storageType == mrtSingular) {
    void **p_targetInst = (void **)
            ((uintptr_t)inst + sourceRole->storageOffset) ;
    MRT_Instance *targetInst = *p_targetInst ;
    *p_targetInst = NULL ; // ❶

    MRT_AssociationRole const *targetRole = &assoc->target ;
    if (targetInst != NULL && targetInst->alloc > 0 &&
            targetInst->classDesc == targetRole->classDesc) { // ❷
        mrtUnlinkBackref(targetRole, inst, targetInst) ;
    }
} else {
    // Simple forward association links are always singular.
    mrtFatalError(mrtRelationshipLinkage) ;
}
```

❶ Assigning `NULL` to the primary reference effectively unlinks the target instance.

❷ We only want to attempt to unlink the back reference if the target instance is still allocated and of the same class. We accomodate the `NULL` case also, since this can arise if the construction of the instance errored out because of a static relationships. Normally, instance don't just change their class out from under themselves. However, union based subclass instances do. Migrating a union subclass instance results in any back reference pointers to the instance remaining valid, but the class of the instance now occupying the memory has changed. Union subclasses are somewhat painful.

The case of unlinking back references for a simple association, occurs when the target instance is being deleted. In this case, we only delete the back references made by the target. In particular we make no attempt to follow the source references and delete the primary reference pointer. If the referring instance is not later deleted, then referential integrity checking will find a pointer to an unallocated instance. This will be caught in referential integrity checking. we must be prepared for the fact that the back references are already gone. This is because it is possible to delete the referenced instance of a simple association before deleting the referring instance.

```
<<mrtDeleteLinks: unlink simple backward>>=
MRT_AssociationRole const *targetRole = &assoc->target ;

void *linkStorage = (void *)((uintptr_t)inst + targetRole->storageOffset) ;
switch (targetRole->storageType) {
case mrtSingular: {
```

```
    void **p_sourceInst = linkStorage ;
    *p_sourceInst = NULL ;                                  // ❶
}
    break ;

case mrtLinkedList: {                                       // ❷
    MRT_LinkRef *sourceList = linkStorage ;                 // ❸
    assert(sourceList->next != NULL && sourceList->prev != NULL) ;
    for (MRT_LinkRef *iter = mrtLinkRefBegin(sourceList) ;
            iter != mrtLinkRefEnd(sourceList) ; ) {
        MRT_LinkRef *sourceInst = iter ;
        iter = iter->next ;                                 // ❹
        mrtLinkRefRemove(sourceInst) ;
    }
}
    break ;

case mrtArray: {
    MRT_ArrayRef * alinks = linkStorage ;                   // ❺
    if (alinks->links != NULL) {
        // Can't unlink array type linkages.
        mrtFatalError(mrtStaticRelationship) ;
    }
}
    break ;

default:
    mrtFatalError(mrtRelationshipLinkage) ;
    break ;
}
```

❶ For the singular pointer case we can just write `NULL` to the back reference. Even if it was already `NULL`, it won't matter. The link is severed.

❷ In this case, the instance is being deleted and all of its back links must be removed.

❸ This is the list of back links to source instances.

❹ Be careful to advance the iterator before deleting the item from the list.

❺ We allow attempting to unlink array backlinks as long as there aren't really any backlinks. This case can arise in creating instances that participate in static relationships and then trying to delete them after catching the fatal error.

In this function we are trying to delete a single back link from the "target" to the "source". This is the back link deletion associated with deleting the primary reference pointer and we have tracked to a target instance.

```
<<mrt forward references>>=
static void
mrtUnlinkBackref(
    MRT_AssociationRole const *const targetRole,
    void *source,
    void *target) ;
```

```
<<mrt static functions>>=
static void
mrtUnlinkBackref(
    MRT_AssociationRole const *const targetRole,
    void *const source,
    void *const target)
{
    assert(source != NULL) ;
```

```
    assert(target != NULL) ;
    assert(targetRole != NULL) ;

    void *linkStorage =
            (void *)((uintptr_t)target + targetRole->storageOffset) ;
    switch (targetRole->storageType) {
    case mrtSingular: {
        void **p_sourceInst = linkStorage ;                              // ❶
        if (*p_sourceInst == source) {                                   // ❷
            *p_sourceInst = NULL ;
        }
    }
        break ;

    case mrtLinkedList: {
        MRT_LinkRef *srcblinks = (MRT_LinkRef *)
                ((uintptr_t)source + targetRole->linkOffset) ;           // ❸
        if (srcblinks->next != NULL && srcblinks->prev != NULL) {        // ❹
            MRT_LinkRef *targetList = linkStorage ;
            for (MRT_LinkRef *targetlink = mrtLinkRefBegin(targetList) ;
                    targetlink != mrtLinkRefEnd(targetList) ;
                    targetlink = targetlink->next) {
                if (targetlink == srcblinks) {
                    mrtLinkRefRemove(srcblinks) ;
                    break ;                                              // ❺
                }
            }
        }
    }
        break ;

    case mrtArray: {
        MRT_ArrayRef *alinks = linkStorage ;
        if (alinks->links != NULL) {                                     // ❻
            // Can't unlink array type linkages.
            mrtFatalError(mrtStaticRelationship) ;
        }
    }
        break ;

    default:
        mrtFatalError(mrtRelationshipLinkage) ;
        break ;
    }
}
```

❶ The entity at the storageOffset is a simple pointer back to the source instance. Point to where the source reference is located in the target instance.

❷ We only want to unlink the backref if it is actually pointing back to the source.

❸ The source instance is linked onto a list whose list terminus is contained in the target instance. We must make sure the source instance is actually linked on the list before we try to remove it from the list. This means running the list to find the match. We only need a pointer to the links member in the source instance to remove it from the list (it is doubly linked). The links are offset into the source instance by the `linkOffset` given in the "target" role relationship descriptor.

❹ Check that the instance is actually linked on the list. If the source has been deleted previously, then it will have been removed from the list already.

❺ Once found and removed, we no longer need to iterate on the rest of the list.

❻ It's alright to unlink a static relationships as long as there were no links to begin with.

For class based associations, we deal with the association in two steps, between each participant and the associator class.

```
<<mrtDeleteLinks: unlink class based association>>=
MRT_ClassAssociation const *assoc = &rel->relInfo.classAssociation ;

if (instclass == assoc->associator.classDesc) {
    MRT_AssociatorRole const *assocRole = &assoc->associator ;
    MRT_AssociationRole const *sourceRole = &assoc->source ;
    MRT_AssociationRole const *targetRole = &assoc->target ;

    void **p_targetInst = (void **)
            ((uintptr_t)inst + assocRole->forwardOffset) ; // ❶
    MRT_Instance *targetInst = *p_targetInst ;
    *p_targetInst = NULL ;
    if (targetInst != NULL && targetInst->alloc > 0 &&
            targetInst->classDesc == targetRole->classDesc) {
        mrtUnlinkBackref(targetRole, inst, targetInst) ;
    }

    void **p_sourceInst = (void **)
            ((uintptr_t)inst + assocRole->backwardOffset) ;
    MRT_Instance *sourceInst = *p_sourceInst ;
    *p_sourceInst = NULL ;
    if (sourceInst != NULL && sourceInst->alloc > 0) {
        mrtUnlinkBackref(sourceRole, inst, sourceInst) ;
    }
}
```

❶ Start with the forward direction to the target. Point to where the target reference is located in the associator instance.

For reference generalizations, the references are singular pointers between the superclass and subclass instances. We must make sure the classes given actually participate in the generalization.

```
<<mrtDeleteLinks: unlink reference generalization>>=
MRT_RefGeneralization const *gen = &rel->relInfo.refGeneralization ;

if (instclass != gen->superclass.classDesc) {
    // Instance is a subclass instance
    int subclassCode = mrtFindRefGenSubclassCode(instclass, gen->subclasses,
            gen->subclassCount) ;
    // Obtain the pointer to the superclass instance.
    void **p_superInst = (void **)
            ((uintptr_t)inst + gen->subclasses[subclassCode].storageOffset) ;
    MRT_Instance *superInst = *p_superInst ;
    *p_superInst = NULL ;
    // NULL out the pointer in the superclass instance pointing to the subclass
    // instance. Watch for a NULL reference to the superclass instances.
    // This can happen if the subclass was simply created on it's on.
    if (superInst != NULL && superInst->alloc > 0) {
        void **p_subInst = (void **)
                ((uintptr_t)superInst + gen->superclass.storageOffset) ;
        *p_subInst = NULL ;
    }
} else {
    // Instance is a superclass instance
    // NULL out the pointer to the subclass instance only,
    // i.e. only the back reference.
    void **p_subInst = (void **)((uintptr_t)inst + gen->superclass.storageOffset) ;
    *p_subInst = NULL ;
}
```

Reclassifying Subclasses

Generalization relationships represent a disjoint union of the subclasses. Because of this property, we know that there is an unconditional relationship between a subclass instance and a superclass instance and an unconditional relationship between a superclass instance and exactly one subclass instance from among all the subclasses of the generalization. This situation leads to the concept of a reclassify operation to allow a related subclass instance to migrate from one subclass to another. It is a powerful concept to model modal operations in a domain. The run-time provides a function to accomplish the details.

```
<<mrt internal external interfaces>>=
extern void *
mrt_Reclassify(
    MRT_Relationship const *rel,
    void *sub,
    MRT_Class const *const newSubclass) ;
```

rel
A pointer to a relationship description for the relationship across which the reclassify operation is to happen.

sub
A pointer to a class instance that is to be reclassified.

newSubclass
A pointer to the class description to which the currently related subclass instance is to be reclassified.

The `mrt_Reclassify` function migrates the class of the `sub` instance along the `rel` generalization to be of the new class, `newSubclass`. Conceptually, reclassification implies that the `sub` instance is unrelated and deleted and a new instance of `newSubclass` is created and related to the superclass instance to which `sub` was previously related.

The return value of the function is an instance pointer to the newly created subclass instance.

Since we support two different ways to store subclass instances, the reclassification considers each type separately.

```
<<mrt external functions>>=
void *
mrt_Reclassify(
    MRT_Relationship const *rel,
    void *sub,
    MRT_Class const *const newSubclass)
{
    assert(rel != NULL) ;
    assert(newSubclass != NULL) ;

    MRT_Instance *currentSubInst = sub ;
    assert(currentSubInst != NULL) ;
    assert(currentSubInst->alloc > 0) ;

    void *newSubInst = NULL ;

    if (rel->relType == mrtRefGeneralization) {
        <<mrt_Reclassify: reclassify reference generalization>>
    } else if (rel->relType == mrtUnionGeneralization) {
        <<mrt_Reclassify: reclassify union generalization>>
    } else {
        mrtFatalError(mrtRelationshipLinkage) ;
    }

    return newSubInst ;
}
```

For reference type generalizations, we must perform the complete delete / create sequence.

```
<<mrt_Reclassify: reclassify reference generalization>>=
MRT_RefGeneralization const *const gen = &rel->relInfo.refGeneralization ;

/*
 * Verify the subclass instance is an instance of that class that
 * is part of the relationship.
 */
int subclassCode = mrtFindRefGenSubclassCode(currentSubInst->classDesc,
        gen->subclasses, gen->subclassCount) ;
/*
 * Fetch the superclass instance via the reference in the subclass instance.
 */
MRT_Instance *super = *(MRT_Instance **)((uintptr_t)sub +
        gen->subclasses[subclassCode].storageOffset) ;
/*
 * Check that the subclass instance is related to a superclass that is
 * the correct one for the relationship.
 */
if (gen->superclass.classDesc != super->classDesc) {
    mrtFatalError(mrtRelationshipLinkage) ;
}
/*
 * Verify that the new subclass is indeed a subclass of the relationship.
 * We don't care about the subtype code and are only using
 * mrtFindRefGenSubclassCode() to validate the new requested subclass.
 */
mrtFindRefGenSubclassCode(newSubclass, gen->subclasses, gen->subclassCount) ;
/*
 * Delete the old subclass instance.  Deleting will cause the subclass
 * instance to be unlinked from the generalization.
 */
mrt_DeleteInstance(sub) ;
/*
 * Create a new instance of the new subclass.
 */
newSubInst = mrt_CreateInstance(newSubclass, MRT_StateCode_IG) ;
/*
 * Create the links to the super class instance.
 */
mrt_CreateSimpleLinks(rel, newSubInst, super, true) ;
```

For generalizations implemented as unions, the reclassification sequence is much simpler. There is no need to delete and create a new instance of the new subclass. We need only transform the currently related instance into the new subclass.

```
<<mrt_Reclassify: reclassify union generalization>>=
MRT_UnionGeneralization const *const gen = &rel->relInfo.unionGeneralization ;
/*
 * Verify the subclass instance is an instance of that class that
 * is part of the relationship. We don't need the subclass code.
 */
MRT_Class const *const subClass = currentSubInst->classDesc ;
mrtFindUnionGenSubclassCode(subClass, gen->subclasses, gen->subclassCount) ;
/*
 * Compute the pointer to the superclass instance.
 */
MRT_Instance *super = (MRT_Instance *)((uintptr_t)currentSubInst -
        gen->superclass.storageOffset) ;
/*
 * Check that the subclass instance is related to a superclass that is
 * the correct one for the relationship.
 */
```

```
if (gen->superclass.classDesc != super->classDesc) {
    mrtFatalError(mrtRelationshipLinkage) ;
}
/*
 * Check that the new subclass is one that is part of this generalization.
 * We don't actually need the subclass code itself.
 */
mrtFindUnionGenSubclassCode(newSubclass, gen->subclasses, gen->subclassCount) ;
/*
 * Clean up any relationship pointers in the currently related instance.
 */
mrtDeleteLinks(subClass->classRels, subClass->relCount, currentSubInst) ;
/*
 * The new instance occupies the same memory as the old one.
 */
newSubInst = currentSubInst ;
/*
 * Set up the memory for the subclass instance according to the new subclass.
 */
mrtInitializeInstance(newSubInst, newSubclass, MRT_StateCode_IG) ;
```

This function is the counterpart for finding a subclass among the subclasses for a union generalization.

```
<<mrt static functions>>=
static int
mrtFindUnionGenSubclassCode(
    MRT_Class const *const subclassClass,
    MRT_Class const *const *subclasses,
    unsigned count)
{
    int subcode ;

    for (subcode = 0 ; subcode < count ; subcode++, subclasses++) {
        if (subclassClass == *subclasses) {
            return subcode ;
        }
    }

    mrtFatalError(mrtRelationshipLinkage) ;
}
```

Managing Execution

In this section, we discuss the rules and policies for managing execution sequencing. There are two means available to domain activities to control the sequencing of execution.

- Invoke an ordinary function.
- Generate an event to the instance of a class.

Not much needs to be said about invoking functions. Control is transferred to the entry point and runs until the function is complete, transferring control back to next statement of the caller. Typically, such functions are organized into those that are associated with the domain as whole, a particular class or the instances of a class. Such organization may be helpful to the programmer, but since they are directly supported by the implementation language, the `micca` run-time does not get involved in mediating them.

The run-time does get involved with those computations that must leave off at some point, waiting for some other action in the system or the external environment, and then resume execution maintaining the past history. This type of execution is implemented as a state machine.

State Machine Rules

Each class that has lifecycle behavior may have a state model associated with it and each instance of that class will have a state variable that allows it to execute as an state machine independent of the other instances of the class. The run-time supports a Moore type state model.

In the Moore formulation of state models, activity code is associated with states and is executed upon entry into a state. This is distinguished from the Mealy formulation where actions are associated with the transitions and are executed upon exiting a state. Much writing and discussion has been wasted attempting to justify one type of state model over another. What we know is they are computationally equivalent, *i.e.* we can prove that there is no problem that you can solve with a Moore machine that cannot also be solved with a Mealy machine and *vice versa*. Whether your application is easier to describe with one type rather than the other is something that you alone may decide. Moore machines are the traditional formulation for Executable UML and they have the simplest implementation structures. What we specifically reject here is any use of hierarchical state models. They are unnecessary and add complication that is not welcome. The power of computation in Executable UML is derived from the interaction of simple state machines each of which is tied to the lifecycle of a particular class. If you have some state model that is large and complicated where you think some other kind of higher order structure is needed, the usual reason is that you have multiple classes masquerading as one and further refinement of your analysis is necessary. That is not usually a welcome answer to the situation, because if the analyst had been able to conceive of a better solution, he/she would probably have done so already. When state models are used to describe the lifecycle behavior of a well defined class (and not sets of classes or the domain as whole) there is no need for more complicated state execution schemes such as hierarchies.

Generally, state activities affect other computations in the domain by updating instance attribute values or by generating events to other instances. The important distinction here is that the application code of the state activities does not deal with actually dispatching the events nor does it control which event is dispatched next.

Event Types

There are three types of events:

1. Transition events that cause transitions in state machines.
2. Polymorphic events that are mapped at runtime across a generalization hierarchy to transition events.
3. Creations events that support asynchronous instance creation.

Creation events are transition events that are also associated with an instance creation. Our strategy for creation events is to create the instance in an inactive state before queuing the event that will activate the instance and cause a transition. We will have need to distinguish between the various event types and use an enumeration to accomplish that.

```
<<mrt internal simple types>>=
typedef enum {
    mrtTransitionEvent,
    mrtPolymorphicEvent,
    mrtCreationEvent
} MRT_EventType ;
```

The process of signaling an event involves the following steps:

1. Obtain an **Event Control Block** (ECB) from the free pool of ECB's.
2. Set the values of the fields in the ECB.
3. Queue the ECB for later dispatch.

Event Control Block

The Event Control Block (ECB) is the primary data structure for signaling and dispatching events.

```
<<mrt internal aggregate types>>=
typedef struct mrtecb {
    struct mrtecb *next ;
    struct mrtecb *prev ;
    MRT_EventCode eventNumber ;
    MRT_AllocStatus alloc ;
    MRT_Instance *targetInst ;
    MRT_Instance *sourceInst ;
    TIMQ_ElementID timer ;
    alignas(max_align_t) MRT_EventParams eventParameters ;
} MRT_ecb ;
```

`next, prev`
: Event queuing is done by doubly linked lists and the links are allocated as part of the ECB as the `next` and `prev` members.

`eventNumber`
: The event number. Events are also encoded as small zero based sequential integers that are unique only within the class to which they are associated. Event numbers are ultimately used as an array index.

`alloc`
: The `alloc` member is yet another part of *event-in-flight* detection. We will discuss that more below when we discuss event dispatch. For now, this member of the ECB holds the value of the allocation counter for the instance that is the target of the event. So when an event is generated, a copy of the current value of the `alloc` member of the instance is stored in the ECB.

`targetInst`
: A pointer to the instance that is to receive the event.

`sourceInst`
: A pointer to the instance that is the signaler of the event. If the event is generated outside the context of an instance (*e.g.* in an domain operation), then this member value is set to `NULL`. The `sourceInst` member also serves an important role in enforcing the rules for delayed events. More on that later.

`timer`
: The identifier of the timer associated with any delay applied to the event before it is dispatched.

`eventParameters`
: Storage for any parameter values passed along with the event. We discuss event parameter data storage below.

Event codes are captured by small integer values.

```
<<mrt interface simple types>>=
typedef uint8_t MRT_EventCode ;
```

Delay times are in milliseconds and we wish to have a wide dynamic range of times.

```
<<mrt interface simple types>>=
typedef uint32_t MRT_DelayTime ;
```

Note that there is no notion of priority contained in the ECB. Some software architectures queue events in a priority order. That is not supported here. Frankly, if you need event priorities to make your system work, then you need to revisit your design or look for a software architecture that supports multiple threads of execution.

Like all the other data structures, there is a storage pool for ECB's and we define a size for it here that can be overridden on the compiler command line. Sizing the pool for ECB's can be difficult. It must be worst case allocation as running out of ECB's is a fatal system error. The pool must be sized to account for the maximum number of events that can be in flight at the same time. This includes delayed events, since they can be considered to be slow flying events.

```
<<mrt interface constants>>=
#ifndef MRT_EVENT_POOL_SIZE
#   define MRT_EVENT_POOL_SIZE 32
#endif /* MRT_EVENT_POOL_SIZE */
```

We must allocate the memory for the ECB storage. As usual, storage is just an array of structures.

```
<<mrt static data>>=
static MRT_ecb mrtECBPool[MRT_EVENT_POOL_SIZE] ;
```

Event Parameter Storage

We need to design how events that carry parameteric data will operate. In this formulation of state machines, events may carry additional parameters. Space has to be allocated for that data. The difficulty is that the parameter data must be given a type. There are a couple of solutions, neither of which is very satisfying. We could collect all the parameters from all the state machines in the system and create a giant union. This would properly allocate the amount of parameter storage required and provide a type safe manner to deal with that data. Unfortunately, the parameters to states are scattered in very many places in a system and gathering them together is a difficult undertaking.

Here we take the approach of providing a fixed amount of memory and letting state activities cast that memory into the appropriate type. Needless to say, this can also be a source of errors, but is much easier to manage. This choice makes sense for many systems. The number of states that use parametric data is usually small and using a fixed size works better than might be expected upon first consideration. The important point here is that events can carry data with them. Many state machine formulations don't support this and it is very difficult to correctly manage memory lifetime without it. It is one of those things that you might not use very often but it is difficult to do without when you need it.

We fix the amount of memory used for event parameter storage, allowing it to be overridden by the defining the appropriate macro.

```
<<mrt interface constants>>=
#ifndef MRT_ECB_PARAM_SIZE
#   define MRT_ECB_PARAM_SIZE  32
#endif /* MRT_ECB_PARAM_SIZE */
```

```
<<mrt interface simple types>>=
typedef char MRT_EventParams[MRT_ECB_PARAM_SIZE] ;
```

The platform model has been very careful to insure the type signatures of events and state activities. When signaling an event, the code generator will generate code to insert event parameters into the data area using the signature associated with the event. In the state activity, the parameters will be moved into local variables according to the signature of the state activity.

Note that the parameters are passed by value but care must be taken when passing pointer references to data (*e.g.* `NUL` terminated strings passed as pointers). The run-time does nothing to manage the life time of the storage when values are passed by reference.

Event Queues

In this run-time architecture, we do asynchronous event dispatch from an event queue. This is one of the simplest ways to insure that we meet the requirement for state activities to run to completion. Since a queue is used, as a state activity executes and potentially signals other events, we know those events will not be dispatched until after the state activity completes. Therefore, there is no danger of a long complicated chain of event dispatching cycling back around to alter the state of the instance or potentially modify some data value that the state activity accesses after generating the event. The guarantee of run-to-completion for state activity execution is very important. We now examine the code that performs the queueing.

To serve as the head of the linked lists, we define an event queue structure that just contains the `next` and `prev` pointers.

```
<<mrt implementation aggregate types>>=
typedef struct mrteventqueue {
    MRT_ecb *next ;
    MRT_ecb *prev ;
} MRT_EventQueue ;
```

There are four queues that are used to manage events.

```
<<mrt static data>>=
static MRT_EventQueue eventQueue ;
static MRT_EventQueue tocEventQueue ;
static MRT_EventQueue delayedEventQueue ;
static MRT_EventQueue freeEventQueue ;
```

The `eventQueue` queue holds events waiting for immediate dispatch. The events in this queue are those signaled as part of an ongoing thread of control. The `tocEventQueue` queue holds events that start a new thread of control. Typically, those are events generated outside of a domain or delayed events finally being delivered. The `delayedEventQueue` queue holds events that are to be delivered by the run-time at some future time. Finally, the `freeEventQueue` queue holds those ECB's that are not currently begin used. From the data structures and the semantics of the queuing, a given ECB can be on at most one of the queues at any time. Most of the time each ECB is on exactly one of the queues, but there are short times when an ECB is not in any queue and a pointer to the ECB is held in a local variable.

The operations that are performed on event queues are associated with adding and removing elements. This code is very conventional and I'm sure you seen it or something very much like it many times before.

```
<<mrt implementation static inlines>>=
static inline MRT_ecb *
mrtEventQueueBegin(
    MRT_EventQueue *queue)
{
    return queue->next ;
}
```

```
<<mrt implementation static inlines>>=
static inline MRT_ecb *
mrtEventQueueEnd(
    MRT_EventQueue *queue)
{
    return (MRT_ecb *)queue ;
}
```

```
<<mrt implementation static inlines>>=
static inline bool
mrtEventQueueEmpty(
    MRT_EventQueue *queue)
{
    return mrtEventQueueBegin(queue) == mrtEventQueueEnd(queue) ;
}
```

```
<<mrt implementation static inlines>>=
static inline void
mrtEventQueueInsert(
    MRT_ecb *item,
    MRT_ecb *at)
{
    item->prev = at->prev ;
    item->next = at ;
    at->prev->next = item ;
    at->prev = item ;
}
```

```
<<mrt implementation static inlines>>=
static inline void
mrtEventQueueRemove(
    MRT_ecb *item)
{
    item->prev->next = item->next ;
    item->next->prev = item->prev ;
    item->prev = item->next = NULL ; // ❶
}
```

❶ Although it is not strictly necessary to NULL out the pointers when an item is removed from the queue, we do depend upon this to know that an item has been removed from a list and can be placed into a different list.

Since we have a pool of ECB's, we need some operations to manage the pool. We start with initialization. This places all the ECB's in the pool onto the free event queue.

```
<<mrt static functions>>=
static void
mrtECBPoolInit(void)
{
    assert(MRT_EVENT_POOL_SIZE >= 1) ;
    /*
     * Initialize the queue terminus structures.
     */
    eventQueue.next = eventQueue.prev = (MRT_ecb *)&eventQueue ;
    tocEventQueue.next = tocEventQueue.prev = (MRT_ecb *)&tocEventQueue ;
    delayedEventQueue.next = delayedEventQueue.prev = (MRT_ecb *)&delayedEventQueue ;
    freeEventQueue.next = freeEventQueue.prev = (MRT_ecb *)&freeEventQueue ;
    /*
     * Place all the event control blocks on the free event
     * queue.  Allocation occurs from there.
     */
    for (MRT_ecb *ecb = mrtECBPool ;
            ecb < mrtECBPool + MRT_EVENT_POOL_SIZE ; ecb++) {
        mrtEventQueueInsert(ecb, mrtEventQueueEnd(&freeEventQueue)) ;
    }
}
```

Event allocation is just removing an ECB from the free list. *N.B.* that running out of Event Control Blocks is fatal.

```
<<mrt static functions>>=
static inline MRT_ecb *
mrtECBalloc(void)
{
    if (mrtEventQueueEmpty(&freeEventQueue)) {
        mrtFatalError(mrtNoECB) ;
    }

    MRT_ecb *ecb = freeEventQueue.next ;
    mrtEventQueueRemove(ecb) ;
    memset(ecb, 0, sizeof(*ecb)) ; // ❶
    return ecb ;
}
```

❶ We zero out the ECB. Although not strictly necessary, it is convenient for debugging, especially when there are event parameters involved.

```
<<mrt static functions>>=
static inline void
mrtECBfree(
    MRT_ecb *ecb)
{
    assert(ecb != NULL) ;

    mrtEventQueueInsert(ecb, mrtEventQueueEnd(&freeEventQueue)) ;
}
```

We will have need to find particular events in a queue. Events are identified by the source of the event, the target of the event and the number of the event.

```
<<mrt static functions>>=
static MRT_ecb *
mrtFindEvent(
    MRT_EventQueue *queue,
    MRT_Instance *sourceInst,
    MRT_Instance *targetInst,
    MRT_EventCode event)
{
    /*
     * Simple iteration through the list of events in the queue.
     */
    for (MRT_ecb *iter = mrtEventQueueBegin(queue) ;
            iter != mrtEventQueueEnd(queue) ;
            iter = iter->next) {
        if (iter->sourceInst == sourceInst && iter->targetInst == targetInst &&
                iter->eventNumber == event) {
            return iter ;
        }
    }
    return NULL ;
}
```

Event Signaling

Signaling an event is a very common operation for a state activity. In this section, we describe the code to accomplish event signaling.

Obtaining An ECB

As we mentioned previously, before an event can be signaled, we must obtain and fill in an ECB. We start our description with a function that does just that.

```
<<mrt internal external interfaces>>=
extern MRT_ecb *
mrt_NewEvent(
    MRT_EventCode event,
    void *target,
    void *source) ;
```

event
The numerical code for the event.

target
A pointer to the instance that is to receive the event.

source
A pointer to the instance that is sending the event. If the event is being sent outside of the context of a class instance, then this argument should be set to `NULL`.

The `mrt_NewEvent` function allocates an ECB structure, fills in the elements and returns the newly minted ECB.

```
<<mrt external functions>>=
MRT_ecb *
mrt_NewEvent(
    MRT_EventCode event,
    void *target,
    void *source)
{
    MRT_Instance *targetInst = target ;
    MRT_Instance *sourceInst = source ;

    assert(targetInst != NULL) ;
    assert(targetInst->alloc != 0) ;
    assert(event < targetInst->classDesc->eventCount) ;

    MRT_ecb *ecb = mrtECBalloc() ;

    ecb->eventNumber = event ;
    ecb->alloc = targetInst->alloc ;     // ❶
    ecb->targetInst = targetInst ;
    ecb->sourceInst = sourceInst ;
    ecb->timer = -1 ;                    // ❷

    return ecb ;
}
```

❶ Note that we initialize the ECB `alloc` member from the target instance. This is an essential part of detecting *event in flight* errors.

❷ Delayed or periodic event signaling will allocate a timer queue element as necessary. Most frequently, we are performing imminent signaling.

Posting an Event

Once you have obtained an ECB initialized for an event, then you need to fill in any event parameter data. Frequently, there are no parameters for an event. Then the ECB is ready to be placed on a queue. There is a distinction between events an instance sends to itself and those that an instance sends to a different instance. Self directed events are placed on the front of the event queue so that they are dispatched in preference to the non-self directed events. This is one of the fundamental state machine

dispatch rules. Posting an event involves determining the correct place in the queue for the ECB. The decision is made based on the ECB values.

```
<<mrt internal external interfaces>>=
extern void
mrt_PostEvent(
    MRT_ecb *ecb) ;
```

ecb
A pointer to an Event Control Block (ECB) that is to be queued for dispatch.

The `mrt_PostEvent` function queues the ECB pointed to by `ecb` to an appropriate event queue.

```
<<mrt external functions>>=
void
mrt_PostEvent(
    MRT_ecb *ecb)
{
    assert(ecb != NULL) ;
    assert(ecb->targetInst != NULL) ;

        /*
         * The location in some event queue where the ECB will be inserted.
         */
    MRT_ecb *qloc ;

    if (ecb->sourceInst == NULL || mrtTransLevel > 0) {          // ❶
        qloc = mrtEventQueueEnd(&tocEventQueue) ;
    } else if (ecb->sourceInst != ecb->targetInst) {             // ❷
        qloc = mrtEventQueueEnd(&eventQueue) ;
    } else {                                                     // ❸
        for (qloc = mrtEventQueueBegin(&eventQueue) ;
                qloc != mrtEventQueueEnd(&eventQueue) &&
                    qloc->sourceInst == qloc->targetInst ;
                qloc = qloc->next) {
            // N.B. -- empty loop
        }
    }

    mrtEventQueueInsert(ecb, qloc) ;
}
```

❶ All events that are signaled outside of a state activity start a new thread of control and the event is queued to the end of the thread of control event queue. If the `sourceInst` is NULL, then the event was signaled directly from a domain operation as this is what the `micca` code generator arranges. If the transaction level is non-zero, then we are in the middle of a synchronous service. This case can arise when a domain operation invokes an instance operation. The source of the event, in that case, is the instance, but the context is still outside of a state activity.

❷ Ordinary transitioning events directed between distinct instances are queued to the end of the event queue.

❸ Self directed events are queued to the front of the event queue, but we have to find the appropriate location. That location is the first event in the queue that is not self directed. The loop preserves the order of event dispatch in the highly unlikely (but not technically illegal) case that a state activity signals multiple self directed events.

One Step Event Signaling

For the case where the event has no parameters or the parameters have already been marshalled into a parameter block, a single function can create the event and post it.

```
<<mrt internal external interfaces>>=
extern void
mrt_SignalEvent(
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst,
    void const *eventparams,
    size_t paramsize) ;
```

event
The numerical code for the event.

targetInst
A pointer to the instance that is to receive the event. If `targetInst` is `NULL`, then no event is signaled.

sourceInst
A pointer to the instance that is sending the event. If the event is being sent outside of the context of a class instance, then this argument should be set to `NULL`.

eventparams
A pointer to the event parameters for the event. If the event requires no additional parameters, then this argument is passed as `NULL`.

paramsize
The number of bytes of event parameter data pointed to by `eventparams`. If the event requires no additional parameters, then this argument is passed as `0`.

```
<<mrt external functions>>=
void
mrt_SignalEvent(
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst,
    void const *eventparams,
    size_t paramsize)
{
    if (targetInst == NULL) {
        return ;                                                    // ❶
    }

    MRT_ecb *ecb = mrt_NewEvent(event, targetInst, sourceInst) ;

    if (eventparams != NULL) {
        assert(paramsize <= sizeof(ecb->eventParameters)) ;
        size_t toCopy = paramsize <= sizeof(ecb->eventParameters) ?
                paramsize : sizeof(ecb->eventParameters) ;
        memcpy(ecb->eventParameters, eventparams, toCopy) ;
    }

    mrt_PostEvent(ecb) ;
}
```

❶ We interpret signaling the `NULL` instance as a request to do nothing. This can be used to avoid tests against `NULL` when the result of a relationship navigation needs to be the target of the event.

Asynchronous Instance Creation

As we mentioned earlier, asynchronous instance creation is accomplished by a combination of creating an instance and then sending it an event. The transition caused by the event then causes a state activity to be executed. The creation is asynchronous in the sense that after an activity requests an asynchronous creation, the return from the request is immediate, but the instance does not come into existance until its creation event is dispatched.

```
<<mrt internal external interfaces>>=
extern void *
mrt_CreateInstanceAsync(
    MRT_Class const *const targetClass,
    MRT_EventCode event,
    void const *eventparams,
    size_t paramsize,
    void *sourceInst) ;
```

targetClass
A pointer to the class description for the instance that is to be created.

event
The number of the event to signal to the newly created instance.

eventparams
A pointer to the event parameters for the creation event.

paramsize
The number of bytes of event parameter data pointed to by `eventparams`.

sourceInst
A pointer to the class instance that is the source of the creation event. If the creation event is signaled outside of a state activity, then this value should be `NULL`.

The `mrt_CreateInstanceAsync` function asynchronously creates an instance of the class described by `targetClass` and arranges for `event` to be signaled to the new instance. Note the event parameters must be provided. After the function returns, the instance has been created in an inactive state and the event has been posted. The return value is the pointer to the newly created instance and can be used to initialize the values of attribute of the instance.

```
<<mrt external functions>>=
void *
mrt_CreateInstanceAsync(
    MRT_Class const *const targetClass,
    MRT_EventCode event,
    void const *eventparams,
    size_t paramsize,
    void *sourceInst)
{
    assert(targetClass != NULL) ;
    assert(targetClass->edb != NULL) ;
    assert(targetClass->edb->creationState >= 0) ;
    assert(event < targetClass->edb->eventCount) ;

    MRT_Instance *targetInst = mrt_CreateInstance(targetClass,
            targetClass->edb->creationState) ;                    // ❶
    targetInst->alloc = -targetInst->alloc ;                      // ❷

    mrt_SignalEvent(event, targetInst, sourceInst, eventparams, paramsize) ;

    return targetInst ;
}
```

❶ We want the instance created in the pseudo-initial state.

❷ The `alloc` member is made negative for instances that are awaiting the dispatch of a creation event. Effectively the negative `alloc` value signals that the memory is reserved but the instance is not yet active. During event dispatch, this value will be turned back into a positive one. Note that the effect here is to use one of the instance slots as a temporary buffer to hold the attribute values for the instance until the creation event is dispatched. This temporary usage must be accounted for when defining the number of instance memory slots for a class.

As we have already seen, subclass instances stored as a union require separate treatment. The following function is the asynchronous counterpart to `mrt_CreateUnionInstance`.

```
<<mrt internal external interfaces>>=
extern void *
mrt_CreateUnionInstanceAsync(
    MRT_Class const *const targetClass,
    MRT_EventCode event,
    void const *eventparams,
    size_t paramsize,
    void *sourceInst,
    MRT_Relationship const *const genRel,
    void *super) ;
```

`targetClass`
A pointer to the class description for the instance that is to be created. The class described by `classDesc` must be a subclass of the relationship described by `genRel`.

`event`
The number of the event to signal to the newly created instance.

`eventparams`
A pointer to the event parameters for the creation event.

`paramsize`
The number of bytes of event parameter data pointed to by `eventparams`.

`sourceInst`
A pointer to the class instance that is the source of the creation event. If the creation event is signaled outside of a state activity, then this value should be `NULL`.

`genRel`
A pointer to a relationship description for the generalization in which the instance is a union-based subclass. The `relType` field of `genRel` must be `mrtUnionGeneralization`.

`super`
A pointer to the superclass instance where the subclass instance is to be created. `super` must be an instance of the superclass described by `genRel`.

The `mrt_CreateUnionInstanceAsync` function asynchronously creates an instance of the union-based subclass described by `targetClass` and arranges for `event` to be signaled to the new instance. Note the event parameters must be provided. After the function returns, the instance has been created in an inactive state and the event has been posted. The return value is the pointer to the newly created instance and can be used to initialize the values of attribute of the instance.

Like the `mrt_CreateUnionInstance` function, the main change here is that the memory for the instance is located in the space allocated for its related superclass instance.

```
<<mrt external functions>>=
void *
mrt_CreateUnionInstanceAsync(
```

```
    MRT_Class const *const targetClass,
    MRT_EventCode event,
    void const *eventparams,
    size_t paramsize,
    void *sourceInst,
    MRT_Relationship const *const genRel,
    void *super)
{
    assert(targetClass != NULL) ;
    assert(targetClass->edb != NULL) ;
    assert(targetClass->edb->creationState >= 0) ;
    assert(event < targetClass->edb->eventCount) ;

    MRT_Instance *targetInst = mrt_CreateUnionInstance(targetClass,
            targetClass->edb->creationState, genRel, super) ;
    targetInst->alloc = -targetInst->alloc ;                              // ❶

    mrt_SignalEvent(event, targetInst, sourceInst, eventparams, paramsize) ;

    return targetInst ;
}
```

❶ As with non-union classes, we mark the instance as allocated but not active. When the creation event is dispatched, the allocation status is changed to indicate the instance is active.

Delayed Events

The concept of a delayed event is to request the run-time to post an event at some time in the future. This implies that the run-time has access to some type of timing facility by which it can know that a given amount of time has elapsed and this implies that the run-time will hold on to the ECB until that future time has arrived. We also interpret delayed events as being signaled from outside of an instance context. So, all delayed events start a new thread of control when they are dispatched.

There is one significant XUML rule associated with delayed events. There can be only one outstanding delayed event of a given event type between any sending / receiving pair of instances (which may be the same instance). This is another way of stating that delayed events are identified by their event name (or numerical encoding), the target instance and the source instance. There are a number of ways to interpret an attempt to generate what amounts to a duplicate delayed event. It could be considered an error, but that is inconvenient and goes against the grain of our attempts to minimize run-time errors. So the run-time regards an attempt to generate a delayed event of the same name between the same sending and receiving pair as a request to cancel the original event and create the new one at its newly given time. This turns out to be very convenient in practice, eliminating the need to perform checks. Cancelling and reinstating a new event turns out to be what is desired in most circumstances.

To understand the implementation of delayed events, it is necessary to understand the way the delayed event queue is maintained. The run-time has a delayed event queue where ECB's are placed awaiting to be posted. In servicing the delayed events, we are particularly trying to avoid doing any periodic computation. For example, we could treat the delayed event queue as a simple list and wake up periodically and run down the list decrementing time values and checking if any events have expired. Such a scheme is easy to implement, but in highly embedded and power sensitive application, periodic activity of this type is wasteful and deemed inappropriate.

In this implementation, we keep the delayed event queue in time relative order. This design meets two important criteria; only a single source of timing is used and there is no periodic execution activity. The cost of meeting these criteria is the price paid to find the appropriate place in the delayed event queue when a delayed event is requested.

There is sometimes a temptation for analysts to try to use delayed events for precise timing control over external interactions. For example, you could conceive of using delayed events as a means of generating a pulse width modulated (PWM) square wave to control a motor. Delayed events are generally **not** suitable for high speed, high precision timing such as that required by motor control. Since we are using only a single timing resource that is shared in the system, we are not able to achieve the precise jitter free timing that would be needed for a PWM. Most microcontrollers have many timing resources available in hardware and they should be used for functions that require high precision. What delayed events are most useful for is time outs for interactions in

the millisecond to minutes range and for those functions where some jitter in the timing is acceptable (*e.g.* blinking an LED). The delayed event time is only guaranteed to be the minimum amount of time that is to elapse before the event is delivered.

There are three functions supplied for dealing with delayed events:

1. Post a delayed event.
2. Cancel a delayed event.
3. Query the remaining time for a delayed event.

The unit of time for delayed events is milliseconds.

Posting A Delayed Event

```
<<mrt internal external interfaces>>=
extern void
mrt_PostDelayedEvent(
    MRT_ecb *ecb,
    MRT_DelayTime time) ;
```

ecb
A pointer to an Event Control Block (ECB) that is to be queued for dispatch.

time
The minimum number of milliseconds of time that must elapse before the event given by `ecb` is posted for dispatch.

The `mrt_PostDelayedEvent` function requests that the event given by `ecb` be dispatched no sooner than `time` milliseconds from now. The value of the `time` argument may be 0, in which case the event is ready for dispatch immediately. All delayed events start a new thread of control. Any request to post a delayed event of the same event number that already exists for some sending / receiving pair of instances results in the first event being canceled and a the event being posted at the given delay time.

```
<<mrt external functions>>=
void
mrt_PostDelayedEvent(
    MRT_ecb *ecb,
    MRT_DelayTime time)
{
    assert(ecb != NULL) ;
    mrtTimeEvent(ecb, time, 0) ;
}
```

The bulk of the work to time the signaling of a event is done by the `mrtTimeEvent` function. This factoring allows us to use the same code for periodic events. One complication in this function is to determine if we are updating an existing delayed event or creating a one. Because of the rules allowing only a single delayed event of a given type between any sending/receiving pair, we interpret an attempt to create a duplicate delayed events as requesting the previous event to be canceled and a new event posted at the newly requested time.

```
<<mrt static functions>>=
static void
mrtTimeEvent(
    MRT_ecb *ecb,
    MRT_DelayTime initial,
    MRT_DelayTime reload)
{
    assert(ecb != NULL) ;
```

```
    TIMQ_TimeTicks initial_ticks = dev_util_timq_ms_to_ticks(initial) ;
    TIMQ_TimeTicks reload_ticks = dev_util_timq_ms_to_ticks(reload) ;

    MRT_ecb *found = mrtFindEvent(&delayedEventQueue,
            ecb->sourceInst, ecb->targetInst, ecb->eventNumber) ;
    if (found == NULL) {
        assert(ecb->timer == -1) ;
        ecb->timer = dev_timq_insert(MRT_TIMER_QUEUE_DEVICE,
                initial_ticks, reload_ticks, (SVC_DevNotifyClosure)ecb) ;
        if (ecb->timer < 0) {
            mrtFatalError(mrtTimerOpFailed, "insert") ;
        }
    } else {
        ecb->timer = found->timer ;                                    // ❶
        mrtEventQueueRemove(found) ;
        mrtECBfree(found) ;

        int status = dev_timq_update(MRT_TIMER_QUEUE_DEVICE, ecb->timer,
                initial_ticks, reload_ticks) ;
        if (status == -ERR_OPERATION_FAILED) {                         // ❷
            ecb->timer = dev_timq_insert(MRT_TIMER_QUEUE_DEVICE,
                    initial_ticks, reload_ticks, (SVC_DevNotifyClosure)ecb) ;
            if (ecb->timer < 0) {
                mrtFatalError(mrtTimerOpFailed, "insert") ;
            }
        } else if (status < 0) {
            mrtFatalError(mrtTimerOpFailed, "update") ;
        }
    }

    mrtEventQueueInsert(ecb, mrtEventQueueEnd(&delayedEventQueue)) ;
}
```

❶ When changing the delay times for an existing delayed event, we remove the old ECB and insert a new one. This handles any difference in event parameters between the old and new events. However, we continue to use the timer ID from the previous event and simply update the time values.

❷ It is possible to loose the race between a delayed event time expiring and attempting to update existing event times, *i.e.* the timing could have expired just before the call to post a delayed event and the notification is still in the background notification queue. In that case, we allocate a new timer for the event.

```
<<mrt internal includes>>=
#include "dev_svc_timq_req.h"
```

We see here the distinction between posting a delayed event and the ordinary posting of an event. For delayed events, the event will end up on the delayed event queue or the thread of control queue. The determining factor is the delay time. If the delay is zero, then the event goes to the thread of control queue directly. Otherwise, it is placed on the thread of control queue after the delay time. Delayed events never go to the ordinary event queue.

There are five main actions for inserting a delayed event.

1. Convert the delay value from millisecond units to units of `ticks`. A tick is platform specific. Computers sometimes don't typically keep time in conventional human units, particularly small embedded systems as we are dealing with. However, we would like to run the delayed event queue in system specific units to avoid as much unnecessary conversion as we can. This conversion will be described below for each supported platform.
2. Stop the timing of the delayed event queue. More on this later but the goal of stopping the delayed event queue timing is to freeze the state of the queue so that we may operate on it.

3. Determine if there is already an event matching the one begin posted. This enforces the rule about not having two delayed events of the same type between the same sending / receiving pair. If one is found then it is removed.
4. Assured of no duplicates, the new event can be inserted into the timing queue.
5. Finally, the timing of the delayed queue is started.

Posting A Periodic Event

```
<<mrt internal external interfaces>>=
extern void
mrt_PostPeriodicEvent(
    MRT_ecb *ecb,
    MRT_DelayTime initial,
    MRT_DelayTime reload) ;
```

ecb
A pointer to an Event Control Block (ECB) that is to be queued for dispatch.

initial
The minimum number of milliseconds of time that must elapse before the event given by `ecb` is first posted for dispatch.

reload
The minimum number of milliseconds of time that must elapse after the first posting of the `ecb` before it is reposted periodically after each `reload` interval.

The `mrt_PostPeriodicEvent` function requests that the event given by `ecb` be dispatched no sooner than `initial` milliseconds from now and that the `ecb` be reposted `reload` milliseconds after that. Reposting in `reload` milliseconds continues until the event is cancelled. The value of the `initial` argument may be 0, in which case the event is posted for dispatch immediately. The value of the `repost` argument must be non-zero if the event is to be treated as a periodic event. Invoking `mrt_PostPeriodicEvent` with a `repost` value of zero results in an delayed event. Like delayed events, all periodic events start a new thread of control. Any request to post a periodic event of the same event number that already exists for some sending / receiving pair of instances results in the first event being canceled and the event being posted at the given delay time.

```
<<mrt external functions>>=
void
mrt_PostPeriodicEvent(
    MRT_ecb *ecb,
    MRT_DelayTime initial,
    MRT_DelayTime reload)
{
    assert(ecb != NULL) ;
    mrtTimeEvent(ecb, initial, reload) ;
}
```

One Step Delayed Event Signaling

Just as for immediate events, delayed events without additional parameters or for which the parameters have been marshalled into a parameter block, a single function can both fill in an ECB and post it to the delayed event queue.

```
<<mrt internal external interfaces>>=
extern void
mrt_SignalDelayedEvent(
    MRT_DelayTime time,
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst,
    void const *eventparams,
    size_t paramsize) ;
```

time
The minimum number of milliseconds of time that must elapse before the event is to be dispatched.

event
The numerical code for the event.

targetInst
A pointer to the instance that is to receive the event.

sourceInst
A pointer to the instance that is sending the event. If the event is being sent outside of the context of a class instance, then this argument should be set to `NULL`.

eventparams
A pointer to the event parameters for the event. If the event requires no additional parameters, then this argument is passed as `NULL`.

paramsize
The number of bytes of event parameter data pointed to by `eventparams`. If the event requires no additional parameters, then this argument is passed as `0`.

```
<<mrt external functions>>=
void
mrt_SignalDelayedEvent(
    MRT_DelayTime time,
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst,
    void const *eventparams,
    size_t paramsize)
{
    MRT_ecb *ecb = mrt_NewEvent(event, targetInst, sourceInst) ;

    if (eventparams != NULL) {
        assert(paramsize <= sizeof(ecb->eventParameters)) ;
        size_t toCopy = paramsize <= sizeof(ecb->eventParameters) ?
                paramsize : sizeof(ecb->eventParameters) ;
        memcpy(ecb->eventParameters, eventparams, toCopy) ;
    }

    mrt_PostDelayedEvent(ecb, time) ;
}
```

One Step Periodic Event Signaling

Periodic events should not carry any additional parameters. It is not possible to change the parameter values between event dispatches and any information used for the parameter values can be available in the model when the event is received.

```
<<mrt internal external interfaces>>=
extern void
mrt_SignalPeriodicEvent(
    MRT_DelayTime initial,
    MRT_DelayTime reload,
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst) ;
```

initial
: The minimum number of milliseconds of time that must elapse before the event given by `ecb` is first posted for dispatch.

reload
: The minimum number of milliseconds of time that must elapse after the first posting of the `ecb` before it is reposted periodically after each `reload` interval.

event
: The numerical code for the event.

targetInst
: A pointer to the instance that is to receive the event.

sourceInst
: A pointer to the instance that is sending the event. If the event is being sent outside of the context of a class instance, then this argument should be set to `NULL`.

```
<<mrt external functions>>=
void
mrt_SignalPeriodicEvent(
    MRT_DelayTime initial,
    MRT_DelayTime reload,
    MRT_EventCode event,
    void *targetInst,
    void *sourceInst)
{
    MRT_ecb *ecb = mrt_NewEvent(event, targetInst, sourceInst) ;

    mrt_PostPeriodicEvent(ecb, initial, reload) ;
}
```

Canceling Delayed Events

Cancelling a delayed event is one of the more complicated delayed event operations. We must account for the various places where a delayed event may be queued.

A delayed event may be in one of two places:

1. In the delayed event queue awaiting either waiting for its time to expire or having already been marked as expired.
2. In the thread of control event queue awaiting dispatch.

We will have more to say about event dispatch below, but it is possible to try to cancel an event after its time has expired but before it has been delivered to the target instance. The mechanisms make the guarantee that after invoking `mrt_EventDelayCancel` the application can be assured that the event will **not** be delivered until such time when it is posted again. Note that it is possible to attempt to cancel a delayed event after it has already been delivered. This is not an error. Unfortunately, the run-time code cannot turn time backwards.

```
<<mrt internal external interfaces>>=
extern void
mrt_CancelDelayedEvent(
    MRT_EventCode event,
    void *target,
    void *source) ;
```

event
The number of the event.

targetInst
A pointer to the instance structure that is to receive the event.

sourceInst
A pointer to the instance structure that is sending the event. Events generated outside of a class instance set this argument to `NULL`.

The `mrt_CancelDelayedEvent` function cancels the delayed event given by `event` and signaled from `source` to `target`. After the return from this function the event is guaranteed not to be delivered.

```
<<mrt external functions>>=
void
mrt_CancelDelayedEvent(
    MRT_EventCode event,
    void *target,
    void *source)
{
    assert(target != NULL) ;

    MRT_ecb *found = mrtRemoveDelayedEvent(event, target, source) ;

    if (found == NULL) {
        found = mrtFindEvent(&tocEventQueue, source, target, event) ;
    }

    if (found != NULL) {
        mrtEventQueueRemove(found) ;
        mrtECBfree(found) ;
    }
}
```

```
<<mrt static functions>>=
static MRT_ecb *
mrtRemoveDelayedEvent(
    MRT_EventCode event,
    void *target,
    void *source)
{
    assert(target != NULL) ;

    MRT_ecb *found =
            mrtFindEvent(&delayedEventQueue, source, target, event) ;
    if (found != NULL) {
        assert(found->timer >= 0) ;
        int status = dev_timq_remove(MRT_TIMER_QUEUE_DEVICE, found->timer) ; // ❶
        assert(status >= 0) ;
        if (status < 0) {
            mrtFatalError(mrtTimerOpFailed, "remove") ;
        }
```

```
    }
    return found ;
}
```

Time Remaining for a Delayed Event

The last provided operation on delayed events is to query the amount of time remaining for a particular delayed event. Since we hold the delayed events in sorted order of time differences, the task of determining the amount of remaining time involves traversing the queue and summing the time increments of all the events in front of the event of interest. The only special case here is what to do if we don't find the delayed event at all. In that case, zero is returned.

```
<<mrt internal external interfaces>>=
extern MRT_DelayTime
mrt_RemainingDelayTime(
    MRT_EventCode event,
    void *target,
    void *source) ;
```

event
The number of the event.

targetInst
A pointer to the instance structure that is to receive the event.

sourceInst
A pointer to the instance structure that is sending the event. Events generated outside of a class instance set this argument to `NULL`.

The `mrt_RemainingDelayTime` function returns the amount of time remaining before the `event` is posted from `source` to `target`. The remaining time will be 0 if the event has already been posted or is such a delayed event does not exist. A return value of 0 does *not* imply that the event has already been dispatched.

The algorithm for computing the remaining time is to simply walk the delayed event queue from the beginning, summing up the set of time delays until we reach the ECB of delayed event.

```
<<mrt external functions>>=
MRT_DelayTime
mrt_RemainingDelayTime(
    MRT_EventCode event,
    void *target,
    void *source)
{
    assert(target != NULL) ;

    TIMQ_TimeTicks remaining = 0 ;                                    // ❶
    MRT_ecb *found =
            mrtFindEvent(&delayedEventQueue, source, target, event) ;
    if (found != NULL) {
        assert(found->timer >= 0) ;
        int status = dev_timq_remaining(MRT_TIMER_QUEUE_DEVICE, found->timer,
                &remaining) ;
        assert(status >= 0) ;
        if (status < 0) {
            mrtFatalError(mrtTimerOpFailed, "remove") ;
        }
    }
```

```
    return (MRT_DelayTime)dev_util_timq_ticks_to_ms(remaining) ;
}
```

❶, ❶ In case of not finding the event, the return is zero.

Note that zero is returned if we did not find the event on the delayed queue. Returning zero does *not* tell us if the event has already been dispatched or might be still in flight on the event queue.

Expired Events in the Delayed Event Queue

When the timer queue device was opened, a background notification proxy function was supplied. This function receives the callback when a background notification for a timer element changes status. In the case of expired events, the primary action taken is to remove the event from the delayed event queue and place it on the TOC event queue. Periodic events pose a special situation. For them, the event is cloned and the clone is posted to the TOC event queue. We must leave the original in the delayed event queue for the next time around. The timer queue device sends the status we need to make that determine as part of the background notification.

```
<<mrt forward references>>=
static void
mrtExpireDelayedEvent(
    SVC_DevTimqNotification const *const notify) ;
```

```
<<mrt static functions>>=
static void
mrtExpireDelayedEvent(
    SVC_DevTimqNotification const *const notify)
{
    MRT_ecb *ecb = (MRT_ecb *)notify->notify_closure ;
    if (ecb->timer == notify->element_id) {                          // ❶
        switch (notify->status) {
        case timq_expired:
            mrtEventQueueRemove(ecb) ;
            ecb->timer = -1 ;
            mrtEventQueueInsert(ecb, mrtEventQueueEnd(&tocEventQueue)) ;
            break ;

        case timq_reloaded: {
            MRT_ecb *rld_ecb = mrt_NewEvent(ecb->eventNumber,
                    ecb->targetInst, ecb->sourceInst) ;
            memcpy(rld_ecb->eventParameters, ecb->eventParameters,
                    sizeof(MRT_EventParams)) ;
            mrtEventQueueInsert(rld_ecb, mrtEventQueueEnd(&tocEventQueue)) ;
        }
            break ;

        case timq_discarded:
            mrtEventQueueRemove(ecb) ;
            ecb->timer = -1 ;
            mrtECBfree(ecb) ;
            mrtFatalError(mrtTimerOpFailed, "discarded event") ;     // ❷
            break ;
        }
    }
}
```

❶ It is possible that no match is found. This can occur if a delayed event is cancelled after it has expired but before the background notification is dispatched.

❷ We never expect the timer queue device to be closed, which would cause any previously queued events to be discarded.

Timing Considerations

With timing being such a common activity in programming, there are many system specific situations that arise in obtaining timing services on any particular platform. When running on top of an operating system, it will provide the necessary timing services. Unfortunately, the interface to those services varies from OS to OS. On bare metal platforms, generally you will have to get timer peripherals and interrupts involved. We will do what we can here to factor away the essential logic from the platform specific, but note that getting delayed event services running on any particular platform will require some additional work.

We will try to make as few assumptions about the available timing services of the platform as we can. Here are the constraints on the timing services:

- The is only a single source of timing. That timing source allows us to specify some time value to it and it will respond with some notification (*e.g.* an interrupt) when the given time has elapsed.
- It is possible to determine how much time remains before an event expires.
- It is not acceptable to execute code periodically that does nothing. This is to account for battery powered devices that cannot afford to wake up and check a timing queue only to find out that there is nothing to do. Activity must be strictly event driven and that implies that we can get positive notification when a time period has elapsed.

Event Dispatch

Finally, we arrive at the point where we can discuss event dispatching. Up until this time, we have been concerned with signaling events, *i.e.* queuing events to be delivered. Now we examine the means by which events are delivered to target instances.

First, we must clarify that despite the fact we have been discussing the *event queue* as if it were a single entity, there are in fact two queues used for dispatching events. The reason for two queues is that we need a way to determine the boundaries of a *thread of control*.

All events that originate outside of a state activity or are the result of a delayed event start a thread of control. The thread of control ends when all the subsequent events that originate from the event starting the thread of control have been dispatched and any activities resulting from the event dispatch has been run. A thread of control is an important concept because its boundaries determine when the data model must be consistent and when we can check the referential integrity.

So our strategy is to use one queue to hold events that start a thread of control and another queue to hold the events that are awaiting dispatch for the ongoing thread of control. When the queue for the ongoing thread of control is empty, then we can start a new thread of control by using events queued to the thread of control queue. When both queues are empty, there is no work to be done and we must wait for subsequent interactions of the system with the outside world.

Dispatching An Event From a Queue

When the run-time needs to dispatch an event it invokes the `mrtDispatchEventFromQueue` function. This function takes as a parameter the queue from which the event is to be dispatched.

```
<<mrt forward references>>=
static bool mrtDispatchEventFromQueue(MRT_EventQueue *queue) ;

<<mrt static functions>>=
static bool
mrtDispatchEventFromQueue(
    MRT_EventQueue *queue)
{
    static MRT_ecb *ecb = NULL ;            // ❶

    if (ecb != NULL) {                      // ❷
        mrtECBfree(ecb) ;
        ecb = NULL ;
    }

    if (!mrtEventQueueEmpty(queue)) {
```

```
        ecb = mrtEventQueueBegin(queue) ;
        mrtEventQueueRemove(ecb) ;
        mrtDispatchEvent(ecb) ;              // ❸
        mrtECBfree(ecb) ;
        ecb = NULL ;

        return true ;
    }

    return false ;
}
```

❶ There is a tricky complication in dispatching an event. It is possible for the event dispatch to cause a fatal error. It is also possible to override the fatal error handler to deal with system specific issues. The overridden fatal error handler can `longjmp` out of the run-time code. Indeed this is the preferred way for testing code. However, if this happens, we could loose the ECB that caused the fatal error. So, while we are dispatching the event, the ECB is stored in a static variable and that static variable is used to indicate the state of completion of the event dispatch.

❷ If the ECB is still around, we presume that we did *not* complete the function after the invocation of `mrtDispatchEvent`. In that case we free our dangling ECB pointer. All event dispatch comes through this function, so a single pointer suffices to save the state of the last dispatch.

❸ Invoking `mrtDispatchEvent` causes transitions in the state machine and can potentially generate a fatal system error.

There are two distinct types of events: transitioning events and polymorphic events. The event types are distinct in that the numerical encoding of the events is used in different ways. In the platform model, all the events for a given state model must have distinct names. The distinction in the platform model between Transitioning Event and Deferred Event is the difference we now see between an ordinary transition event and a polymorphic event. The code generator will assign a unique integer to each event. The set of transition events will be encoded sequentially starting at 0 up to E - 1, where E is the number of transitioning events in the state model. Polymorphic events for the class are then encoded starting at E up to E + P - 1, where P is the number of polymorphic events. Note that only superclasses that have both polymorphic events (either defined or inherited) and a state model of its own will have both transition events and polymorphic events defined for it. However, all events have a unique name in the platform model and a unique numerical encoding in the run-time.

Encoding the event number in this fashion makes it convenient to identify an event for a class when, for example, it is in a delayed event queue. As we will see below, at dispatch time, any polymorphic event numbers will have the number of transition events for the class (E in the discussion above) subtracted from their encoded number so that they can be used as an array index into the data structures used for polymorphic event mapping.

There is a secondary usage of transition events for asynchronous instance creation. A transition event that is dispatched when the current state of the instance is the pseudo-initial creation state is deemed a creation event. We wish to treat creation events separately because of the design strategy we have used to implement asynchronous instance creation. The strategy creates the instance in the creation state, marks its memory slot as allocated but inactive, and queues a transition event. When the event is dispatched, the instance memory slot must be marked active and a transition event dispatch happens. By marking the instance as inactive, but allocated, we can make sure the instance data is not accessed during the time between when it was created and when the creation event is actually dispatched. But, because there is additional processing to be done before a creation event is dispatched as a transition event, we have to be able to identify a transition event as being used in an asynchronous instance creation context.

We provide separate functions to dispatch each event type.

```
<<mrt forward references>>=
static void mrtDispatchTransitionEvent(MRT_ecb *ecb) ;
static void mrtDispatchPolymorphicEvent(MRT_ecb *ecb) ;
static void mrtDispatchCreationEvent(MRT_ecb *ecb) ;
```

Once an event has been chosen for dispatch, the logic to determine the type of the event and, therefore, which dispatch function to invoke is shown below.

```
<<mrt forward references>>=
static void mrtDispatchEvent(MRT_ecb *ecb) ;
```

```
<<mrt static functions>>=
static void
mrtDispatchEvent(
    MRT_ecb *ecb)
{
    assert(ecb != NULL) ;

    MRT_Instance *targetInst = ecb->targetInst ;
    assert(targetInst != NULL) ;

    MRT_Class const *const classDesc = targetInst->classDesc ;
    assert(classDesc != NULL) ;

    MRT_edb const *const edb = classDesc->edb ;
    if (edb == NULL) {
        mrtDispatchPolymorphicEvent(ecb) ;                          // ❶
    } else {
        if (ecb->eventNumber < edb->eventCount) {                   // ❷
            if (targetInst->currentState == edb->creationState) {   // ❸
                mrtDispatchCreationEvent(ecb) ;
            } else {
                mrtDispatchTransitionEvent(ecb) ;
            }
        } else {
            assert(classDesc->pdb != NULL) ;
            assert(ecb->eventNumber - edb->eventCount <
                    classDesc->pdb->eventCount) ;
            mrtDispatchPolymorphicEvent(ecb) ;                      // ❹
        }
    }
}
```

❶ If there is no event dispatch block for the class, then all the events of the class must be polymorphic.

❷ If a class has both transitioning events and polymorphic events, the code generator insures that the transitioning events are always numbered starting from 0.

❸ A creation event is just a transitioning event dispatched when we are in the creation state.

❹ If a class has both transitioning events and polymorphic events, the polymorphic events are numbered starting at the end of the sequence for transitioning events.

Event typing is discovered by first determining if there are any transition events. If not, then we must have a polymorphic event (and consequently, its numbering started at 0). Otherwise, if the event number falls in the range of transition events, then it must be either an ordinary transition event or a transition event used as a creation event. Finally, for the case where there are both transition and polymorphic events defined for the class, events outside of the transition event range must be dispatched as polymorphic events.

In the sections below, we consider the dispatch of each particular event type.

Transition Event Dispatch

The most frequent event type to dispatch is that of a transitioning event to a state machine. We refer to these event types as, *transition*, only to distinguish them from the more complicated and less frequent used polymorphic and creation event types.

Dispatching an event in its simplest terms involves using the current state of the instance and the event number contained in an ECB as indices into the transition matrix. The transition matrix is the same for all instances of a class. The entry in the transition matrix is the new state to which a transition is to be made. There are a few additional rules needed to account for *ignored* and *can't happen* events.

Ignored events (IG) cause no transition. Events that are ignored can be thought of as an optimization on the state transition graph. Ignored events could be handled by adding a new state to which the ignored event makes a transition and that new state has all the other outbound transitions that the original state had. Clearly, having the concept of ignored events saves much clutter in the state transition graph and in the implementation of the state transition matrix.

When the analyst considers a transition to be a logical impossibility, then it is declared as a can't happen (CH) event. In the micca run-time, a can't happen transition is treated as a fatal system error. This is a policy decision of the architecture, so **do not** assume that *can't happen* means *shouldn't happen* or *has a low probability of happening*. In this architecture, can't happen means absolutely impossible to happen and if it does happen then there has been a tear in the logic / space / time continuum and the only course available is to give up and declare a fatal error.

The data structure used for transitioning event dispatch is called an **Event Dispatch Block** (EDB). The code generator supplies an EDB for each class (and assigner) that has a state model. Below we will see how all this ties together. For now, we discuss the data structure and how it is used.

```
<<mrt internal aggregate types>>=
typedef struct mrteventdispatchblock {
    MRT_DispatchCount stateCount ;
    MRT_DispatchCount eventCount ;
    MRT_StateCode initialState ;
    MRT_StateCode creationState ;
    MRT_StateCode const *transitionTable ;
    MRT_PtrActivityFunction const *activityTable ;
    bool const *finalStates ;

#       ifndef MRT_NO_NAMES
    char const *const *stateNames ;
#       endif /* MRT_NO_NAMES */
} MRT_edb ;
```

`stateCount`
: The number of states in the transition matrix.

`eventCount`
: The number of events in the transition matrix.

`initialState`
: The number of the state that is the default state when an instance is created synchronously.

`creationState`
: The number of the state that is the default state when an instance is created asynchronously.

`transitionTable`
: A pointer to the transition matrix.

`activityTable`
: A pointer to the state activities.

`finalStates`
: A pointer to a boolean array that determines if a state is a final state.

`stateNames`
: A pointer to an array of character pointers to the names of the class states. This information is used in tracing event dispatch.

The dimensions of the state transition matrix are `stateCount` rows by `eventCount` columns. The counts are held as small integers.

```
<<mrt internal simple types>>=
typedef uint8_t MRT_DispatchCount ;
```

The transition table is in state major order, *i.e.* the current state is used to index conceptual rows and the event number is used to index conceptual columns. The dimensions of the transition table are captured in the EDB to allow run time bounds checking during event dispatch.

The basic transition algorithm is to use the current state of an instance and the event number of an event as the indices into the transition matrix. The entry in the transition matrix is the new state. Notice the very simple data structures required for Moore state machines.

The new state is used as an index into the `activityTable`. The activity table is an array of function pointers to the activity associated with each state.

```
<<mrt internal simple types>>=
typedef void MRT_ActivityFunction(void *const, void const *const) ;
typedef MRT_ActivityFunction *MRT_PtrActivityFunction ;
```

Since Moore state machines associate the activity with the state, that code segment is supplied as a function matching the prototype above. The first argument is a pointer to the instance receiving the event. It is `void` typed and state activities are expected to recover the correct type by casting the pointer to the be of the proper class data structure. The second argument is a pointer to the event parameters. Again, the correct type is recovered in the state activity to match the parameter signature of the activity. Notice that assigning back into event parameters does not make any sense as the parameter values are discarded after the state activity completes.

One other feature of the state machine dispatch rules regards final states. A state may be marked as final and if so, then the run-time will destroy the instance when the state activity is completed. The `finalStates` member points to an array, indexed by state number, that specifies if a particular state is indeed a final state. As is frequently the case, the class may have no final states. In this case, `finalState` member may be set to `NULL` to indicate this fact and save the storage of the final state booleans (*i.e.* there is no need to have an array of `false` values).

```
<<mrt static functions>>=
static void
mrtDispatchTransitionEvent(
    MRT_ecb *ecb)
{
    MRT_Instance *const targetInst = ecb->targetInst ;
    MRT_edb const *const edb = targetInst->classDesc->edb ;
    assert(edb != NULL) ;
    assert(edb->stateCount > targetInst->currentState) ;
    assert(edb->eventCount > ecb->eventNumber) ;
    /*
     * Check for the "event-in-flight" error. This occurs when an instance is
     * deleted while there is an event for that instance in the event queue.
     * For this architecture, such occurrences are considered as run-time
     * detected analysis errors.
     */
    if (targetInst->alloc != ecb->alloc) {
        mrtEventInFlightError(ecb->sourceInst, ecb->eventNumber, targetInst) ;
    }
    /*
     * Fetch the new state from the transition table.
     */
    MRT_StateCode newState = *(edb->transitionTable +
            targetInst->currentState * edb->eventCount + ecb->eventNumber) ;

#       ifndef MRT_NO_TRACE
    /*
     * Trace the transition.
     */
    mrtTraceTransitionEvent(ecb->eventNumber, ecb->sourceInst,
```

```
            ecb->targetInst, targetInst->currentState, newState) ;
#       endif /* MRT_NO_TRACE */

    /*
     * Check for a can't happen transition.
     */
    if (newState == MRT_StateCode_CH) {
        mrtCantHappenError(targetInst, targetInst->currentState,
                ecb->eventNumber) ;
    } else if (newState != MRT_StateCode_IG) {
        assert(newState < edb->stateCount) ;
        /*
         * We update the current state to reflect the transition before
         * executing the activity for the state.
         */
        targetInst->currentState = newState ;
        /*
         * Invoke the state activity if there is one.
         */
        MRT_PtrActivityFunction activity = edb->activityTable[newState] ;
        if (activity) {
            activity(targetInst, &ecb->eventParameters) ;
        }
        /*
         * Check if we have entered a final state. If so, the instance is
         * deleted.
         */
        if (edb->finalStates && edb->finalStates[newState]) {
            mrt_DeleteInstance(targetInst) ; // ❶
        }
    }
}
```

❶ This is where asynchronous instance deletion occurs. If the state is designated as final, then the run-time deletes the instance after its state activity completes.

The processing for dispatching a transition event follows directly from the definitions. After the check for an event-in-flight error, we perform the indexing into the transition matrix. The indexing expression results from the need to treat a linear set of bytes as a two dimensional matrix. We can't type it any differently since we have different sized transition matrices for each different state model. After obtaining the new state, we must determine if we are actually going to make a transition or if the event is to be ignored or considered a fatal error. Assuming that we are transitioning, then the associated state activity is found and executed. Note that empty state activity may be dispensed with and a `NULL` inserted into the action table. After the action, we check if the instance entered a final state.

We are finally in a position to explain the event-in-flight error in detail. Only one analysis error is detected at run-time, the delivery of an event to an instance that has been deleted. Because events are queued, it is possible for an event to be generated for an instance and then while the event is on the queue awaiting to be delivered, the target instance is deleted by some other code executing. For a single threaded architecture, this is considered an analysis error. Deliverying events to deleted instances should never happen! The analytical model is responsible for insuring that instance deletion is accomplished only after there are no events awaiting to be delivered. However, it can happen and the run-time detects and catches this.

A significant difficulty arises in systems that use distinct memory pools for the instances of each class. If an instance is destroyed and another one created, they may very well end up in exactly the same array slot and therefore have exactly the same instance pointer value. So, a pathological case where an event is generated for an instance, the instance is deleted and then re-created while the event is queued could end up delivering the event to the newly recreated instance. Quite the wrong thing to do.

The strategy used here is to vary the number in the `alloc` field of the instance each time it is allocated. Then a copy of the `alloc` field is placed in the ECB when the event is queued. In effect, the real identifier of an instance, for event dispatch purposes, is its pointer plus the value of the `alloc` field. The `alloc` field serves as a *generation* indicator to distinguish one lifetime of an instance from another. When dispatched, the values of the `alloc` fields in the two structures must match or else

the target instance has been destroyed and re-created in the same memory slot. Of course, the observant reader will have seen that in the case where the target instance is destroyed and recreated 16,000 times while the event is queued will result in the event being dispatched to the wrong instance. This is considered such a remote possibility as to be of no practical concern.

Polymorphic Event Dispatch

Polymorphic events in their full generality can be complex, but they are based on a simple idea. In fact, there is nothing going on in the dispatch of polymorphic events that could not otherwise be handled in the state activity. Strictly speaking, polymorphic events must be considered an optimization, but a very convenient and significant one. Previously, we have described the rules concerning polymorphic events.

When a polymorphic event is dispatched, we must traverse the generalization from the superclass to the subclass to determine the type of the subclass. Conceptually, determining which subclass is related to a particular instance of a superclass is not difficult. There are two fundamental steps in dispatching a polymorphic event:

1. Determining which subtype instance is currently related to a supertype instance.
2. Mapping the polymorphic event encoding in the supertype to an event encoding in the subtype.

In order to accomplish the first step, the run-time has know how the generalization relationship is stored in the instances. The run-time supports two different schemes of storing the generalization relationship, either as a pointer reference or as a union member of the superclass structure.

Both relationship storage techniques have their uses and we will not discuss the pros and cons of one choice over another here. If the generalization relationship is stored as a reference, then the superclass instances will contain an instance pointer to a subtype. If the generalization relationship is stored as a union, then the superclass structure will a member that is a union of the data types of all the subtypes of the generalization hierarchy.

As we will see, if we can locate where in the superclass instance structure the subclass encoding and the subclass reference or union are located, then we can determine the type of the subclass instance to which a particular superclass instance is related. To do that we assume that the there is a data type that can hold the byte offset from the beginning of the superclass instance structure to the required information. As you can probably imagine, this will be a tricky piece of code since it must pick out information from an arbitrary data structure in a generic fashion.

```
<<mrt internal simple types>>=
typedef size_t MRT_AttrOffset ;
```

Polymorphic Event Mapping

We now turn our attention to the actual mapping of polymorphic events. The mapping is analogous to the mapping of current state and event to a new state for normal event dispatch. For polymorphic events, the mapping is from subclass code of the currently related subclass and polymorphic event number to a new event. The data structure required for this is given by:

```
<<mrt internal aggregate types>>=
typedef struct mrtgendispatchblock {
    struct mrtrelationship const *relship ;
    MRT_EventCode const *eventMap ;
} MRT_gdb ;
```

`relship`
: A pointer to the relationship description for the relationship across which the polymorphic event will be dispatched.

`eventMap`
: The `eventMap` member is a pointer to the mapping of polymorphic events for the generalization. This mapping is indexed in major order by subclass code and in minor order by polymorphic event number.

A key realization here is that as we are mapping along a generalization relationship, a given polymorphic event may be mapped into a normal event where it will be consumed by the state machine of the class or it may be delegated further down the hierarchy. To be delegated further implies that a polymorphic event will be mapped into yet another polymorphic event to be further mapped in a subsequent dispatch.

Now we can tie it all together. For a superclass class that has associated polymorphic events the code generator supplies a Polymorphic Dispatch Block (PDB) to direct the run-time as to how to perform the mapping of polymorphic events to transitioning events.

```
<<mrt internal aggregate types>>=
typedef struct mrtpolydispatchblock {
    MRT_DispatchCount eventCount ;
    MRT_DispatchCount genCount ;
    struct mrtgendispatchblock const *genDispatch ;

#       ifndef MRT_NO_NAMES
    char const *const *genNames ;
#       endif /* MRT_NO_NAMES */
} MRT_pdb ;
```

eventCount
: The `eventCount` member holds the number of polymorphic events associated with the superclass. Like transition events, polymorphic events are encoded as zero based sequential integers so they may be used as array indices in the mapping process.

genCount
: The `genCount` member holds the number of generalization that originate at the superclass class.

genDispatch
: The `genDispatch` member holds a pointer to an array of Generalization Dispatch Blocks. The array contains `genCount` elements.

genNames
: A pointer to an array of generalization relationship names. The array is of length, `genCount`.

Now we can give the code for polymorphic event dispatch.

```
<<mrt static functions>>=
static void
mrtDispatchPolymorphicEvent(
    MRT_ecb *ecb)
{
    MRT_Instance *superInst = ecb->targetInst ;
    MRT_Class const *const superClassDesc = superInst->classDesc ;
    MRT_pdb const *const pdb = superClassDesc->pdb ;
    assert(pdb != NULL) ;
    assert(pdb->genCount > 0) ;
    /*
     * Check for the "event-in-flight" error. We must make sure the
     * superclass instance still exists.
     */
    if (superInst->alloc != ecb->alloc) {
        mrtEventInFlightError(ecb->sourceInst, ecb->eventNumber, superInst) ;
    }
    /*
     * Compute the base offset for polymorphic event numbering.  This base
     * offset will be used to turn the polymorphic event number into an array
     * index.
     */
    MRT_edb const *const edb = superClassDesc->edb ;
    MRT_EventCode eventOffset = edb == NULL ? 0 : edb->eventCount ;
```

```
    assert(ecb->eventNumber >= eventOffset) ;
    assert(ecb->eventNumber - eventOffset < pdb->eventCount) ;
    /*
     * Save the original event number.  We intend to reuse the same ECB for each
     * event we dispatch and will need this and the super class instance pointer
     * values should there be more than one generalization associated with this
     * superclass.
     */
    MRT_EventCode origEvent = ecb->eventNumber ;
    /*
     * For each generalization that originates at the superclass an event is
     * generated down that generalization to one of the subclasss.
     */
    MRT_gdb const *gdb = pdb->genDispatch ;
    for (unsigned gnum = 0 ; gnum < pdb->genCount ; gdb++, gnum++) {
        MRT_Relationship const *const rel = gdb->relship ;
        MRT_Instance *subInst ;
        int subclassCode ;

        /*
         * Find the target instance reference and the class of the target
         * instance. How we do this depends upon how the generalization is
         * stored in the superclass instance.
         */
        if (rel->relType == mrtRefGeneralization) {
            /*
             * When the generalization is implemented via a pointer, we need an
             * extra level of indirection to fetch the address of the subclass.
             */
            MRT_RefGeneralization const *gen = &rel->relInfo.refGeneralization ;
            subInst = *(MRT_Instance **)
                    ((uintptr_t)superInst + gen->superclass.storageOffset) ;
            assert(subInst != NULL) ;
            /*
             * We must also guard against the possibility that the subclass was
             * unrelated from the superclass before the polymorphic event was
             * dispatched.
             */
            if (subInst == NULL) {
                mrtFatalError(mrtRelationshipLinkage) ;
            }
            assert(subInst->classDesc != NULL) ;
            subclassCode = mrtFindRefGenSubclassCode(subInst->classDesc,
                    gen->subclasses, gen->subclassCount) ;
        } else if (rel->relType == mrtUnionGeneralization) {
            /*
             * When the generalization is implemented by a union, we need only
             * point to the address of the subclass since it is contained
             * within the superclass.
             */
            MRT_UnionGeneralization const *gen =
                    &rel->relInfo.unionGeneralization ;
            subInst = (MRT_Instance *)
                    ((uintptr_t)superInst + gen->superclass.storageOffset) ;
            assert(subInst->classDesc != NULL) ;
            subclassCode = mrtFindUnionGenSubclassCode(subInst->classDesc,
                    gen->subclasses, gen->subclassCount) ;
        } else {
            mrtFatalError(mrtRelationshipLinkage) ;
        }
        /*
         * Check that our subclass instance is indeed allocated and usable.  We
```

```
         * are trying to guard against the possiblity that the subclass
         * instance was deleted before the polymorphic event was delivered.
         */
        assert(subInst->alloc > 0) ;
        if (subInst->alloc <= 0) {
            mrtEventInFlightError(ecb->sourceInst, origEvent, subInst) ;
        }
        /*
         * Update the target and allocation status in the ECB to match
         * that of the subclass instance, which is where the event
         * is now directed.
         */
        ecb->targetInst = subInst ;
        ecb->alloc = subInst->alloc ;
        /*
         * Fetch the event number for the subclass from the polymorphic
         * mapping.  The class of the subclass related to the superclass
         * determines the mapped value for the event.  Note we must subtract
         * off any offset in the event encoding that was consumed by the
         * transition events.
         */
        ecb->eventNumber = *(gdb->eventMap + subclassCode * pdb->eventCount +
                origEvent - eventOffset) ;

#           ifndef MRT_NO_TRACE
        /*
         * Trace the transition.
         */
        mrtTracePolymorphicEvent(origEvent, ecb->sourceInst, superInst,
                subclassCode, gnum, ecb->eventNumber) ;
#           endif /* MRT_NO_TRACE */

        mrtDispatchEvent(ecb) ; // ❶
    }
}
```

❶ Since a polymorphic event may map to either another polymorphic event or to a transition event, we use the general dispatch function to recursively dispatch the mapped event.

The code loops through all of the generalizations for which the `superInst` is a superclass. The vast majority of the time there is only one generalization. The strategy is to reuse the ECB that was carrying the polymorphic event as the ECB for the remapped events. This saves allocating a new ECB and avoids any problem that there may not be an ECB available at that time. The mapping of a polymorphic event is a function of the superclass target and the polymorphic event number. So we hold them in local variables as we overwrite the necessary fields in the ECB to hold the mapped event information.

The core of the algorithm is to determine the subclass code of the related subclass instance and use that as the row index into the polymorphic event map for the generalization. The event number (appropriately offset by the number of transition events) is then used as the column index to find the mapping entry. That mapping entry contains a new event number. The new target of the event is the currently related subclass instance. As we have discussed, this may be a stored as a pointer or may be a union member of the superclass instance structure. For the pointer case, we fetch the pointer from its location in the superclass structure. For the union case, the location in the superclass structure is the beginning of the union, *i.e.* we down cast to the subclass member. We fill in the `alloc` field to enable the event in flight detection just in case the mapped event causes a transition. Finally the newly minted ECB is recursively dispatched and the next generalization is considered. Recursively dispatching the event preserves the order of delivery of the events. Note that when there are multiple generalization hierarchies for the event, the dispatch order is determined by the order the code generator decides for generalization dispatch blocks.

Creation Event Dispatch

Fortunately, creation events are much simpler than polymorphic events. Creation event dispatch fixes up the `alloc` field of the target and the ECB before normal event dispatch. No additional data structures are required.

```
<<mrt static functions>>=
static void
mrtDispatchCreationEvent(
    MRT_ecb *ecb)
{
#       ifndef MRT_NO_TRACE
    /*
     * Trace the transition.
     */
    mrtTraceCreationEvent(ecb->eventNumber, ecb->sourceInst, ecb->targetInst,
            ecb->targetInst->classDesc) ;
#       endif /* MRT_NO_TRACE */

    assert(ecb->alloc == ecb->targetInst->alloc) ;
    assert(ecb->alloc < 0) ;
    assert(ecb->targetInst->alloc < 0) ;
    assert(ecb->targetInst->currentState ==
        ecb->targetInst->classDesc->edb->creationState) ;

    ecb->alloc = ecb->targetInst->alloc = -ecb->targetInst->alloc ; // ❶
    mrtDispatchTransitionEvent(ecb) ;
}
```

❶ By negating the `alloc` value, we show that the instance is active.

Bridging Domains

All but the simplest of systems will contain more that one domain. A domain represents a coherent subject matter with its own set of rules and policies. Domains are also the unit of encapsulation and reuse.

Consider a simple example of a domain that controls a chemical reaction vessel. One aspect of synthesizing something in a reaction vessel is controlling the temperature of the vessel. Such systems typically delegate control to another domain so that a Reaction Management domain would delegate setting and maintaining the reaction vessel temperature to a Signal I/O domain.

From the point of view of Reaction Management, the reaction vessel has heaters, coolers, pumps, valves and such things and it is its responsibility to sequence the vessel operations to accomplish the synthesis. Reaction Management does not know the details of what it is controlling. It wants to set the vessel temperature to 57C and maintain it there for some time period. How that happens in terms of heaters, thermal load and other such physical considerations is delegated.

From the point of view of Signal I/O, it knows that it is controlling output actuators with input sensors as feed back. That output point number 57 corresponds to a reactor vessel is not in its scope of concern.

The concept of a domain is very much related to the concept of *separation of concerns*. But that which is separated must be combined if we are to obtain a useful system. The difference in the semantic view points of domains must be bridged by translating the assumptions and dependencies of one domain into the services provided by other domains.

There are several approaches to bridging. The most elegant approach is *implicit bridgeing* which is related to *aspect oriented programming* concepts. In this style of bridging, we would use a separate means to describe how operations in one domain would be mapped to side effects in another domain. For example, we would want to be able to state that when the state activity in the Reaction Management domain updates the reactor vessel set point attribute we would want the temperature to be transferred to output point number 57 of the Signal I/O domain. The advantage of this type of bridging is that the individual domains are not modified and the domain semantics are not disturbed. The bridge interactions are defined outside of the domain itself. It is then a code generator's task to generate the domain code so that the bridge mapping is implemented. The disadvantage of this type of bridging is that it is very difficult to implement both the means to specify the domain interactions and the code generator required to generate the bridging code.

Another approach to bridging is termed, *explicit bridging*. In this form, a domain makes explicit invocations to an *external entity*. The invocations demonstrate the explicit requirements that the domain delegates and the service it requires. Bridges then

map the external entity invocations to interactions with other domains that provide the required service. Explicit bridging is a workable technique but can lead to a large number of domain operations being needed to service a client domain's dependencies. Often, a client domain and service domain will have *counterpart classes*. These are classes that represent different aspects of the same entity but do so in the semantics of the individual domains. In such cases, model level operations in the client domain, *e.g.* updating an attribute value, are bridged to a similar model operations in the service domain, *e.g* updating the counterpart class attribute with a scaled value. If it is necessary to code a domain operation for each such model level operation, the service domain will be unnecessarily complicated and its reuse will be limited since the details of how model level actions need to take place will very much depend upon the context in which the service domain is used.

Making available the ability to do some model level operations such as updating attributes or signaling events from outside of the domain can make explicit bridging much easier and prevent cluttering the external interfaces of service domains with domain operations specifically built to support the use of the service domain in one particular system. Doing so breaks the encapsulation of the domain, albeit in a very controlled manner.

`Micca` takes the following view of bridging:

- Domain designs should provide domain operations for those computations that cannot be accomplished by a single simple model level operation. Operations such as navigating a relationship or other more complicated activities of instance creation and relationship linkage require processing code to accomplish and that code is best gathered in a domain operation.
- The `micca` code generator provides the means to break encapsulation of a domain for simple model level actions such as signaling an event or reading a class instance attribute.
- A domain must be populated before it can be bridged. If there is an initial instance population, bridges usually must account for that. If the population of instances for a class is static, this will usually greatly simplify the bridge.
- Bridge operation code is manually coded to map the semantics of one domain onto another and, in general, is specific to the use of the bridged domains in a particular system and, therfore, *not* reusable.
- Dynamic activity in one domain may have to be reflected in the bridge. For example, instance creation in one domain may have to result in creating an instance of a counterpart class in a service domain. In such cases, the bridge itself may have to track the dynamics of the two domains.

These consideration make domain bridging abstract and complicated. Sadly, the methodological fundamentals of bridging seem to be lacking and much of what is done to build systems from bridged domains smacks of being *ad hoc*. There is much more that could be said about bridging that space does not allow for here. Projects are cautioned that bridging domains can be a significant activity for which plans need to be made.

In this section, we discuss the facilities provided by `micca` to perform operations on a domain from outside the domain itself. These facilities are intended to be used by bridge operation code. Conceptually, `micca` builds a *portal* into the domain and supplies a set of functions that will reach through the portal and perform operations in the domain. The set of operations that can be performed is very limited but is a useful set for implementing bridge operations.

Portal Data Structures

The only externally scoped identifiers in the generated "C" code file are those of the domain operations. All other generated code contains identifiers that are file static in scope. This is done to prevent collisions and contention of names at link time. As a result, other translation units have no access to the internals of a domain as was the design goal.

Since we intend to provide some operations on the internals of a domain we need a way to identify the entities upon which we are operating. Internally, a `micca` generated domain uses pointers to refer to instances, class descriptors and other essential data. We do *not* want to expose addresses of the internals of a domain to the outside. Since all class instances are really elements of the class storage array, the index of an instance in the storage array serves as a convenient external identifier. The `micca` code generator will place a set of "C" preprocessor definitions in the generated header file to numerically encode the classes, instances and attributes. It is these small integer numbers that will serve as identifiers outside of the domain and which we will use as array indices (with appropriate bounds checking) inside the domain.

```
<<mrt interface simple types>>=
typedef unsigned short MRT_ClassId ;
typedef unsigned short MRT_InstId ;
```

```
typedef unsigned short MRT_AttrId ;
typedef size_t MRT_AttrSize ;
typedef unsigned short MRT_AssignerId ;
```

To operate on the internals of a domain requires a data structure that maps the numerically encoded identifiers to the internals of domain.

```
<<mrt interface aggregate types>>=
struct mrtdomainportal ;
typedef struct mrtdomainportal MRT_DomainPortal ;

<<mrt internal aggregate types>>=
struct mrtdomainportal {
    unsigned classCount ;
    MRT_Class const *classes ;
    unsigned assignerCount ;
    MRT_Class const *assigners ;

#       ifndef MRT_NO_NAMES
    char const *name ;
#       endif /* MRT_NO_NAMES */
} ;
```

`classCount`
: The number of classes in the domain.

`classes`
: A pointer to an array of class descriptions for the domain. The array contains `classCount` elements.

`assignerCount`
: The number of assigners in the domain.

`assigners`
: A pointer to an array of class description for the assigners. The array contains `assignerCount` elements.

`name`
: A pointer to a NUL terminated character array giving the name of the domain.

The domain portal is a collection of class descriptions for the classes and assigners in the domain. Note that relationships are not accessible via the portal.

For each domain, the code generator will create an externally scoped variable of the above type. The name of the variable follows the form `<domain name>__PORTAL`, *i.e.* the suffix `__PORTAL` is appended to the domain name. This variable forms the *portal* into the domain and below we describe the operations on the domain that are available via the portal.

Portal Access Functions

In this section we describe the set of operations that are available through the portal.

Portal Errors

All the portal access functions return an integer value. Non-negative return values indicate success and return the requested information. Negative return values indicate errors as described below.

```
<<mrt interface constants>>=
    // No such class.
#define MICCA_PORTAL_NO_CLASS       (-1)
    // No such instance.
#define MICCA_PORTAL_NO_INST        (-2)
```

```
    // No such attribute.
#define MICCA_PORTAL_NO_ATTR        (-3)
    // Instance slot is not in use.
#define MICCA_PORTAL_UNALLOC        (-4)
    // Class does not have a state model.
#define MICCA_PORTAL_NO_STATE_MODEL (-5)
    // No such event for the class.
#define MICCA_PORTAL_NO_EVENT       (-6)
    // No such state for the class.
#define MICCA_PORTAL_NO_STATE       (-7)
    // Class does not support dynamic instances.
#define MICCA_PORTAL_NO_DYNAMIC     (-8)
    // Operation not allowed on a dependent attribute.
#define MICCA_PORTAL_DEPENDENT_ATTR (-9)
    // Operation failed from insufficient space to transfer value.
#define MICCA_PORTAL_SIZE_ERROR     (-10)
```

It is useful to have a function to translate the error code number into a canonical string for the error.

```
<<mrt external interfaces>>=
extern char const *
mrt_PortalErrorString(
    int portalErrorCode) ;
```

portalErrorCode
A negative or zero value as returned from a portal function.

The `mrt_PortalErrorString` function translates a portal error code to a human readable string. By special dispensation, a value of zero for `portalErrorCode` is accepted and a string reference is returned for it. The value of `NULL` is returned for unknown error code values or if the run time was compiled with the pre-processor macro, `MRT_NO_NAMES`, defined.

```
<<mrt external functions>>=
char const *
mrt_PortalErrorString(
    int portalErrorCode)
{
#       ifndef MRT_NO_NAMES

    static char const * const portalErrStrings[] = {
        "No error",                                     // ❶
        "No such class",
        "No such instance",
        "No such attribute",
        "Instance slot is not in use",
        "Class does not have a state model",
        "No such event for the class",
        "No such state for the class",
        "Class does not support dynamic instances",
        "Operation not allowed on a dependent attribute",
        "Operation failed from insufficient space to transfer value",
    } ;

    assert(portalErrorCode >= MICCA_PORTAL_SIZE_ERROR && portalErrorCode <= 0) ;

    if (portalErrorCode < MICCA_PORTAL_SIZE_ERROR || portalErrorCode > 0) {
        return NULL ;
    }
```

```
    return portalErrStrings[-portalErrorCode] ;          // ❷

#       else

    return NULL ;

#       endif /* MRT_NO_NAMES */
}
```

❶ We'll allow zero as an error code since it is used to indicate success by several of the portal functions. So we need a place for the success message in the string array.

❷ Negating the error code turns it into an index. We have already validated the argument value so this is safe to do.

References to Attributes

Internally, the portal access functions often need to find a class instance. We have factored that into a static function.

```
<<mrt static functions>>=
static int
mrtPortalGetInstRef(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_Instance **ref)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const pclass = portal->classes + classId ;
    if (instId >= pclass->instCount) {
        return MICCA_PORTAL_NO_INST ;
    }

    MRT_Instance *instance = mrtIndexToInstance(pclass, instId) ;
    if (instance == NULL) {
        return MICCA_PORTAL_UNALLOC ;
    }

    if (instance->classDesc != pclass) {
        return MICCA_PORTAL_NO_INST ;                    // ❶
    }

    if (ref) {
        *ref = instance ;
    }
    return 0 ;
}
```

❶ Normally, the class descriptor pointer in the instance matches that obtained from the portal classes array because we used the class descriptor to convert from an index to an instance reference. However, for union subclasses, it is possible to give an instance id that maps to a storage slot in the superclass storage which currently contains an instance of a different subtype class. This test checks for that circumstance.

There is one case where we expose some internal pointer information and this is for attributes. Attributes implemented as arrays don't pass well by value in "C". Obtaining a reference to the attribute is often the best way to deal with passing its value around.

```
<<mrt external interfaces>>=
extern int
mrt_PortalGetAttrRef(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void **pref,
    MRT_AttrSize *size) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

attrId
The number of the attribute to be read. This number is generated by micca and placed in the domain header file.

pref
A pointer to a memory pointer where the attribute reference is placed. If this argument is `NULL` then no reference is returned.

size
A pointer to where the size of the attribute is placed. If this argument is `NULL` then no size information is returned.

`mrt_GetAttrRef` obtains a pointer to the storage location for an attribute and the size of that storage location. The return value is 0, if the attribute storage could be found. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalGetAttrRef(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void **pref,
    MRT_AttrSize *size)
{
    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const class = instref->classDesc ;
    assert(class != NULL) ;
    if (attrId >= class->attrCount) {
        return MICCA_PORTAL_NO_ATTR ;
    }

    MRT_Attribute const *attr = class->classAttrs + attrId ;
    if (attr->type != mrtIndependentAttr) {
        return MICCA_PORTAL_DEPENDENT_ATTR ;
    }
```

```
    if (pref) {
        *pref = (void *)((uintptr_t)instref + attr->access.offset) ;
    }
    if (size) {
        *size = attr->size ;
    }
    return 0 ;
}
```

Reading Attributes

Instance attribute values may be read through the portal.

```
<<mrt external interfaces>>=
extern int
mrt_PortalReadAttr(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void *dst,
    MRT_AttrSize dstSize) ;
```

portal
: A pointer to the portal structure for the domain.

classId
: The number of the class. This number is generated by micca and placed in the domain header file.

instId
: The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

attrId
: The number of the attribute to be read. This number is generated by micca and placed in the domain header file.

dst
: A pointer to memory where the attribute value is placed.

dstSize
: The number of bytes pointed to by `dst`.

`mrt_PortalReadAttr` reads an attribute value from a domain. No more than `dstSize` bytes will be placed in the memory pointed to by `dst`. If the return value is non-negative, then it represents the actual number of bytes read. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalReadAttr(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void *dst,
    MRT_AttrSize dstSize)
{
    assert(dst != NULL) ;
```

```
    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const class = instref->classDesc ;
    if (attrId >= class->attrCount) {
        return MICCA_PORTAL_NO_ATTR ;
    }

    MRT_Attribute const *attr = class->classAttrs + attrId ;
    MRT_AttrSize srcSize = attr->size ;
    if (srcSize > dstSize) {
        return MICCA_PORTAL_SIZE_ERROR ;
    }

    if (attr->type == mrtIndependentAttr) {
        void *src = (void *)((uintptr_t)instref + attr->access.offset) ;
        memcpy(dst, src, srcSize) ;
    } else {
        attr->access.formula(instref, dst, srcSize) ;
    }
    return srcSize ;
}
```

Updating Attributes

Instance attribute values may be updated through the portal.

```
<<mrt external interfaces>>=
extern int
mrt_PortalUpdateAttr(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void const *src,
    MRT_AttrSize srcSize) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

attrId
The number of the attribute to be updated. This number is generated by micca and placed in the domain header file.

src
A pointer to memory from where the attribute value is taken.

srcSize
The number of bytes pointed to by `src`.

`mrt_PortalUpdateAttr` updates an attribute value in a domain. If the return value is non-negative, then it represents the actual number of bytes copied into the attribute storage location. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalUpdateAttr(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_AttrId attrId,
    void const *src,
    MRT_AttrSize srcSize)
{
    assert(src != NULL) ;

    void *dst ;
    MRT_AttrSize dstSize ;

    int result = mrt_PortalGetAttrRef(portal, classId, instId, attrId, &dst,
            &dstSize) ;
    if (result != 0) {
        return result ;
    }

    if (srcSize > dstSize) {
        return MICCA_PORTAL_SIZE_ERROR ;
    }

    assert(dst != NULL) ;
    memcpy(dst, src, srcSize) ;
```

```
    return srcSize ;
}
```

Signaling Events

The portal allows signalling an event to a class instance.

```
<<mrt external interfaces>>=
extern int
mrt_PortalSignalEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize) ;
```

portal
: A pointer to the portal structure for the domain.

classId
: The number of the class. This number is generated by micca and placed in the domain header file.

instId
: The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

eventNumber
: The event number of the event to signal.

eventParameters
: A pointer to the supplemental event parameters. This may be set to `NULL` if there are not parameters for the event.

paramSize
: The number of bytes pointed to by `eventParameters`.

`mrt_PortalSignalEvent` signals an ordinary or polymorphic event to the given instance. The return value is 0 upon success. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalSignalEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize)
{
    int result ;
    MRT_ecb *ecb ;

    result = mrtPortalNewECB(portal, classId, instId, eventNumber,
            eventParameters, paramSize, &ecb) ;
    if (result == 0) {
        mrt_PostEvent(ecb) ;
    }
```

```
    return result ;
}
```

Internally, we factor obtaining an ECB into its own function.

```
<<mrt static functions>>=
static int
mrtPortalNewECB(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize,
    MRT_ecb **ecbRef)
{
    assert(ecbRef != NULL) ;

    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const classDesc = instref->classDesc ;
    assert(classDesc != NULL) ;
    if (classDesc->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= classDesc->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }

    if (paramSize > sizeof(MRT_EventParams)) {
        return MICCA_PORTAL_SIZE_ERROR ;
    }

    MRT_ecb *ecb = mrt_NewEvent(eventNumber, instref, NULL) ;
    if (eventParameters != NULL) {
        memcpy(ecb->eventParameters, eventParameters, paramSize) ;
    }
    *ecbRef = ecb ;
    return 0 ;
}
```

Signaling Delayed Events

Signaling delayed events is similar to immediate events, but adds a delay time argument.

```
<<mrt external interfaces>>=
extern int
mrt_PortalSignalDelayedEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize,
    MRT_DelayTime delay) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

eventNumber
The event number of the event to signal.

eventParameters
A pointer to the supplemental event parameters. This may be set to `NULL` if there are not parameters for the event.

paramSize
The number of bytes pointed to by `eventParameters`.

delay
The minimum number of milliseconds to delay before the event is dispatched.

`mrt_PortalSignalDelayedEvent` signals an ordinary or polymorphic event to the deliverd to an instance after a delay time. The return value is 0 upon success. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalSignalDelayedEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize,
    MRT_DelayTime delay)
{
    MRT_ecb *ecb ;
    int result = mrtPortalNewECB(portal, classId, instId, eventNumber,
            eventParameters, paramSize, &ecb) ;
    if (result == 0) {
        mrt_PostDelayedEvent(ecb, delay) ;
    }

    return result ;
}
```

Canceling Delayed Events

A delayed event may be cancelled.

```
<<mrt external interfaces>>=
extern int
mrt_PortalCancelDelayedEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

eventNumber
The event number of the event to signal.

`mrt_PortalCancelDelayedEvent` cancels a delayed event. The return value is 0 upon success. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalCancelDelayedEvent(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber)
{
    MRT_Instance *instref ;

    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    assert(instref != NULL) ;
    MRT_Class const *const class = instref->classDesc ;
    assert(class != NULL) ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= class->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }

    mrt_CancelDelayedEvent(eventNumber, instref, NULL) ;
    return 0 ;
}
```

Remaining Delay Time

You may also request the time remaining for a delayed event.

```
<<mrt external interfaces>>=
extern int
mrt_PortalRemainingDelayTime(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    MRT_DelayTime *delayRef) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

eventNumber
The event number of the event to query.

delayRef
A pointer to a location where the remaining delay time is placed.

The `mrt_PortalCancelDelayedEvent` function queries the amount of time remaining before the event with `eventNumber` is dispatched to the `instId` instance of class, `classId`. The value is returned indirectly via the `delayRef` pointer. The returned delay value is zero if the delay time has already elapsed or if no matching delayed event could be found. The return value of the function is zero if successful and the value pointed to by `delayRef` is valid.

```
<<mrt external functions>>=
int
mrt_PortalRemainingDelayTime(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    MRT_DelayTime *delayRef)
{
    assert(delayRef != NULL) ;

    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const class = instref->classDesc ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= class->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }
```

```
    MRT_DelayTime delay = mrt_RemainingDelayTime(eventNumber, instref, NULL) ;
    if (delayRef) {
        *delayRef = delay ;
    }
    return 0 ;

}
```

Synchronous Instance Creation

```
<<mrt external interfaces>>=
extern int
mrt_PortalCreateInstance(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_StateCode initialState) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

initialState
The number of the state into which the new created instance is placed. If the class given by `classId` has no state model, then `initialState` must be given as `MRT_StateCode_IG`. If the class given by `classId` has as state model, then if `initialState` is given as `MRT_StateCode_IG`, the instance will be placed in the default initial state as it was defined when the state model was configured. Otherwise, `initialState` must be a valid state number for the class and the new instance is created in that state. N.B. no state activity is executed in any case.

```
<<mrt external functions>>=
int
mrt_PortalCreateInstance(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_StateCode initialState)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->containment != NULL) {
        return MICCA_PORTAL_NO_DYNAMIC ;
    }

    if (initialState != MRT_StateCode_IG) {
        if (class->edb == NULL) {
            return MICCA_PORTAL_NO_STATE_MODEL ;
        }
        if (initialState <= MRT_StateCode_CH ||
                initialState >= class->edb->stateCount ||
                initialState == class->edb->creationState) {
            return MICCA_PORTAL_NO_STATE ;
        }
    }
```

```
    void *inst = mrt_CreateInstance(class, initialState) ;
    return mrt_InstanceIndex(inst) ;
}
```

Asynchronous Instance Creation

```
<<mrt external interfaces>>=
extern int
mrt_PortalCreateInstanceAsync(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize) ;
```

`portal`
A pointer to the portal structure for the domain.

`classId`
The number of the class. This number is generated by micca and placed in the domain header file.

`eventNumber`
The number of the creation event.

`eventParameters`
A pointer to the parameters for the event. If the event has no supplemental parameters, this argument should be `NULL`.

`paramSize`
The number of bytes pointed to by `eventParameters`.

The `mrt_PortalCreateInstanceAsync` function sends `eventNumber` as a creation event to the class given by `classId`. Any parameters of the event are pointed to by `eventParameters` and are `paramSize` bytes in length. The newly created instance is placed in its pseudo-initial state and the `eventNumber` event is signaled to it. N.B. a transition occurs when the event is delivered resulting in the execution of a state activity.

```
<<mrt external functions>>=
int
mrt_PortalCreateInstanceAsync(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->containment != NULL) {                    // ❶
        return MICCA_PORTAL_NO_DYNAMIC ;
    }

    if (class->edb == NULL) {                            // ❷
```

```
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= class->edb->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }

    MRT_Instance *inst = mrt_CreateInstanceAsync(class,
            eventNumber, eventParameters, paramSize, NULL) ;
    return mrt_InstanceIndex(inst) ;
}
```

❶ Union subclass instances cannot be created asynchronously.

❷ Asynchronous creation requires an event dispatch block.

Deleting Instances

```
<<mrt external interfaces>>=
extern int
mrt_PortalDeleteInstance(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

The `mrt_PortalDeleteInstance` deletes the `instId` instance of the `classId` class.

```
<<mrt external functions>>=
int
mrt_PortalDeleteInstance(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId)
{
    MRT_Instance *inst = NULL ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &inst) ;
    if (result == 0) {
        assert(inst != NULL) ;
        mrt_DeleteInstance(inst) ;
    }

    return result ;
}
```

Signaling Events To Assigners

The portal allows signalling an event to an assigner instance.

```
<<mrt external interfaces>>=
extern int
mrt_PortalSignalEventToAssigner(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize) ;
```

`portal`
A pointer to the portal structure for the domain.

`assignerId`
The number of the assigner. This number is generated by micca and placed in the domain header file.

`instId`
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class. Single assigners have only one instance and so `instId` must be 0 for them. Multiple assigners may have an `instId` value greater than 0.

`eventNumber`
The event number of the event to signal.

`eventParameters`
A pointer to the supplemental event parameters. This may be set to `NULL` if there are not parameters for the event.

`paramSize`
The number of bytes pointed to by `eventParameters`.

`mrt_PortalSignalEventAssigner` signals a transitioning event to the given assigner instance. Polymorphic and creation events cannot be signaled to an assigner. The return value is 0 upon success. Negative numbers indicate an error occurred.

```
<<mrt external functions>>=
int
mrt_PortalSignalEventToAssigner(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId,
    MRT_EventCode eventNumber,
    void const *eventParameters,
    size_t paramSize)
{
    MRT_Instance *instref = NULL ;
    int result = mrtPortalGetAssignerRef(portal, assignerId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    assert(instref != NULL) ;
    MRT_Class const *const asnClass = instref->classDesc ;
    MRT_edb const *edb = asnClass->edb ;
    assert(edb != NULL) ;
    if (eventNumber >= edb->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
```

```
    }

    if (paramSize > sizeof(MRT_EventParams)) {
        return MICCA_PORTAL_SIZE_ERROR ;
    }

    MRT_ecb *ecb = mrt_NewEvent(eventNumber, instref, NULL) ;
    if (eventParameters != NULL) {
        memcpy(ecb->eventParameters, eventParameters, paramSize) ;
    }
    mrt_PostEvent(ecb) ;
    return 0 ;
}
```

Similar to obtaining an instance reference, we factor out code to obtain a reference to an assigner.

```
<<mrt static functions>>=
static int
mrtPortalGetAssignerRef(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId,
    MRT_Instance **ref)
{
    assert(portal != NULL) ;

    if (assignerId >= portal->assignerCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const asnClass = portal->assigners + assignerId ;
    if (instId >= asnClass->instCount) {
        return MICCA_PORTAL_NO_INST ;
    }

    MRT_iab *iab = asnClass->iab ;
    assert(iab != NULL) ;
    MRT_Instance *instref = (MRT_Instance *)
            ((uintptr_t)iab->storageStart + iab->instanceSize * instId) ;
    if (instref->alloc <= 0) {
        return MICCA_PORTAL_UNALLOC ;
    }

    if (ref) {
        *ref = instref ;
    }
    return 0 ;
}
```

Obtaining Class Current State

The portal supports reading the current state of an instance of a class having a state model.

```
<<mrt external interfaces>>=
extern int
mrt_PortalInstanceCurrentState(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId) ;
```

portal
A pointer to the portal structure for the domain.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

instId
The number of the instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the class.

The `mrt_PortalInstanceCurrentState` function returns the number of the current state of the `instId` instance of `classId` class.

```
<<mrt external functions>>=
int
mrt_PortalInstanceCurrentState(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_InstId instId)
{
    MRT_Instance *instref ;
    int result = mrtPortalGetInstRef(portal, classId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    MRT_Class const *const class = instref->classDesc ;
    assert(class != NULL) ;
    if (class->edb == NULL) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    return instref->currentState ;
}
```

Obtaining Assigner Current State

The portal supports reading the current state of an assigner of a class having a state model.

```
<<mrt external interfaces>>=
extern int
mrt_PortalAssignerCurrentState(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId) ;
```

portal
A pointer to the portal structure for the domain.

assignerId
The number of the assigner. This number is generated by micca and placed in the domain header file.

instId
The number of the assigner instance. Instance numbers are consecutive non-negative integers up to the maximum number of instances defined for the assigner.

The `mrt_PortalAssignerCurrentState` function returns the number of the current state of the `instId` instance of `assignerId` assigner.

```
<<mrt external functions>>=
int
mrt_PortalAssignerCurrentState(
    MRT_DomainPortal const *const portal,
    MRT_AssignerId assignerId,
    MRT_InstId instId)
{
    MRT_Instance *instref = NULL ;
    int result = mrtPortalGetAssignerRef(portal, assignerId, instId, &instref) ;
    if (result != 0) {
        return result ;
    }

    assert(instref != NULL) ;
    return instref->currentState ;
}
```

Domain Introspection

For testing and other purposes, it is useful to be able to inquire about the characteristics of a domain at run-time. The functions in this section expose other domain portal meta-information.

```
<<mrt external interfaces>>=
extern char const *
mrt_PortalDomainName(
    MRT_DomainPortal const *const portal) ;
```

portal
A pointer to a domain portal data structure.

The `mrt_PortalDomainName` function returns a pointer to a `NUL` terminated string giving the name of the domain described by the portal data pointed to by `portal`. If the domain was compiled with the preprocessor symbol `MRT_NO_NAMES` defined, then the return value is `NULL`.

```
<<mrt external functions>>=
```

```
char const *
mrt_PortalDomainName(
    MRT_DomainPortal const *const portal)
{
    assert(portal != NULL) ;

#       ifndef MRT_NO_NAMES
    return portal->name ;
#       else
    return NULL ;
#       endif /* MRT_NO_NAMES */
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalDomainClassCount(
    MRT_DomainPortal const *const portal) ;
```

portal
A pointer to a domain portal data structure.

The function `mrt_PortalDomainClassCount` returns the number of classes in the domain. Classes may be identified to the other portal functions using integer numbers ranging from 0 to the return value of `mrt_PortalDomainClassCount` minus 1.

```
<<mrt external functions>>=
int
mrt_PortalDomainClassCount(
    MRT_DomainPortal const *const portal)
{
    assert(portal != NULL) ;

    return portal != NULL ? portal->classCount : 0 ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    char const ** const nameRef) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

nameRef
A pointer to a character pointer where a reference to the class name is placed.

The `mrt_PortalClassName` function obtains the name of the *classId* class in the domain described by *portal*. The class name is returned via `nameRef` as a pointer to a `NUL` terminated string. If the domain was compiled with the preprocessor symbol `MRT_NO_NAMES` defined, then the value placed in the location pointed to by *nameRef* is `NULL`. If successful, the return value of the function is zero. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    char const ** const nameRef)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

#       ifndef MRT_NO_NAMES
    MRT_Class const *const class = portal->classes + classId ;
    if (nameRef) {
        *nameRef = class->name ;
    }
#       else
    if (nameRef) {
        *nameRef = NULL ;
    }
#       endif /* MRT_NO_NAMES */

    return 0 ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassAttributeCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassAttributeCount` function returns the number of attributes in the class identified by *classId* in the domain described by *portal*. If successful, the return value of the function is non-negative and represents the number of attributes of the class. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassAttributeCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    return class->attrCount ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassInstanceCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassInstanceCount` function returns the maximum number of instances that may be created for the class identified by *classId* in the domain described by *portal*. If successful, the return value of the function is positive and represents the number of instance storage slots for the class. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassInstanceCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    return class->instCount ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassEventCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassEventCount` function returns the number of events in the state model of the class identified by *classId* in the domain described by *portal*. If successful, the return value of the function is positive and represents the number of events defined for the state model of the class. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassEventCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId)
{
    assert(portal != NULL) ;
```

```
    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    return class->eventCount ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassStateCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassStateCount` function returns the number of states in the state model of the class identified by *classId* in the domain described by *portal*. If successful, the return value of the function is positive and represents the number of states defined for the state model of the class. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassStateCount(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    MRT_edb const *edb = class->edb ;
    assert(edb != NULL) ;

    return edb != NULL ? edb->stateCount : 0 ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassAttributeName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_AttrId attrId,
    char const ** const nameRef) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

attrId
The number of the attribute. This number is generated by micca and placed in the domain header file.

nameRef
A pointer to a character pointer where a reference to the attribute name is placed.

The `mrt_PortalClassAttributeName` function obtains the name of the *attrId* attribute in the *classId* class in the domain described by *portal*. A pointer to a `NUL` terminated string is placed in the object pointed to by `nameRef`. If the run-time code was compiled with `MRT_NO_NAMES` defined, then the value placed in `nameRef` is `NULL`. If successful, the return value of the function is zero. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassAttributeName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_AttrId attrId,
    char const ** const nameRef)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (attrId >= class->attrCount) {
        return MICCA_PORTAL_NO_ATTR ;
    }

    if (nameRef) {
#           ifndef MRT_NO_NAMES
        *nameRef = class->classAttrs[attrId].name ;
#           else
        *nameRef = NULL ;
#           endif /* MRT_NO_NAMES */
    }

    return 0 ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassAttributeSize(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_AttrId attrId) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

attrId
The number of the attribute. This number is generated by micca and placed in the domain header file.

The `mrt_PortalClassAttributeSize` function returns the number of bytes occupied by the *attrId* attribute in the *classId* of the domain described by *portal*. If successful, the return value of the function is positive and represents the number of bytes of storage allocated to the attribute. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassAttributeSize(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_AttrId attrId)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (attrId >= class->attrCount) {
        return MICCA_PORTAL_NO_ATTR ;
    }

    return class->classAttrs[attrId].size ;
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassEventName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_EventCode eventNumber,
    char const ** const nameRef) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

eventNumber
The number of the event. This number is generated by micca and placed in the domain header file.

nameRef
A pointer to a character pointer where a reference to the event name is placed.

The `mrt_PortalClassEventName` function obtains the name of the *eventNumber* event of the state model in the *classId* class in the domain described by *portal*. A pointer to a `NUL` terminated string is placed in the object pointed to by `nameRef`. If the run-time code was compiled with `MRT_NO_NAMES` defined, then the value placed in `nameRef` is `NULL`. If successful, the return value of the function is zero. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassEventName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_EventCode eventNumber,
    char const ** const nameRef)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    if (class->eventCount == 0) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (eventNumber >= class->eventCount) {
        return MICCA_PORTAL_NO_EVENT ;
    }

#       ifndef MRT_NO_NAMES
    char const *const *eventNames = class->eventNames ;
    if (eventNames != NULL && nameRef != NULL) {
        *nameRef = eventNames[eventNumber] ;
    }
#       else
    if (nameRef != NULL) {
        *nameRef = NULL ;
    }
#       endif /* MRT_NO_NAMES */

    return 0 ;
```

```
}
```

```
<<mrt external interfaces>>=
extern int
mrt_PortalClassStateName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_StateCode stateCode,
    char const **nameRef) ;
```

portal
A pointer to a domain portal data structure.

classId
The number of the class. This number is generated by micca and placed in the domain header file.

stateCode
The number of the state. This number is generated by micca and placed in the domain header file.

nameRef
A pointer to a character pointer where a reference to the state name is placed.

The `mrt_PortalClassStateName` function obtains the name of the *stateCode* state of the state model in the *classId* class in the domain described by *portal*. A pointer to a `NUL` terminated string is placed in the object pointed to by `nameRef`. If the run-time code was compiled with `MRT_NO_NAMES` defined, then the value placed in `nameRef` is `NULL`. If successful, the return value of the function is zero. Upon failure, the return value is one of the portal error codes.

```
<<mrt external functions>>=
int
mrt_PortalClassStateName(
    MRT_DomainPortal const *const portal,
    MRT_ClassId classId,
    MRT_StateCode stateCode,
    char const **nameRef)
{
    assert(portal != NULL) ;

    if (classId >= portal->classCount) {
        return MICCA_PORTAL_NO_CLASS ;
    }

    MRT_Class const *const class = portal->classes + classId ;
    MRT_edb const *edb = class->edb ;
    if (edb == NULL) {
        return MICCA_PORTAL_NO_STATE_MODEL ;
    }

    if (stateCode < 0 || stateCode >= edb->stateCount) {
        return MICCA_PORTAL_NO_STATE ;
    }

    if (nameRef) {
#           ifndef MRT_NO_NAMES
        *nameRef = edb->stateNames[stateCode] ;
#           else
        *nameRef = NULL ;
#           endif /* MRT_NO_NAMES */
    }
    return 0 ;
```

```
}
```

Event Dispatch Tracing

Debugging event driven, callback, state machine based applications can be rather more complicated than conventional, linear flow code. When class instances generate events to other instances and it can be hard to determine the exact sequence of execution by simply examining the source code. Indeed, part of the intent here is to factor away from the application the details of sequencing execution. Setting a breakpoint in the action of a state is easy enough, the difficulties arise when trying to determine where to set a breakpoint to catch the results of the next event dispatch. Given that many events will be flying around a given program, it is very useful to be able to collect the set of event dispatches in chronological order.

To help in debugging, the run-time can be conditionally compiled to support tracing the event dispatch. After the code is properly compiled, a pointer to a trace callback function may be registered and then each event dispatched will result in the function being called with the information about the dispatch.

It should be noted that dealing with trace information can be very difficult. For small embedded systems, there may not be sufficient space to store the strings that give names to states and events (the `MRT_NO_NAMES` macro can be defined to remove the string information). This means that only numbers are available to the tracing code and there is substantial effort required to back translate the numbers into strings that are meaningful to a human.

Because tracing also affects the performance of the run-time code, it may also be excluded by defining the `MRT_NO_TRACE` macro. Most projects will want to define both `MRT_NO_NAMES` and `MRT_NO_TRACE` in code compiled for a release. This, along with defining `NDEBUG`, will result in significantly less initialized data space usage in the executables.

Trace Information

Since there are three types of events, there are three distinct sets of information generated when an event is dispatched. There is information common to all events and information specific to each event type.

```
<<mrt interface trace aggregate types>>=
struct mrttraceinfo ;
typedef struct mrttraceinfo MRT_TraceInfo ;

<<mrt internal trace aggregate types>>=
struct mrttraceinfo {
    <<common trace fields>>
    union {
        <<transition trace fields>>
        <<polymorphic trace fields>>
        <<creation trace fields>>
    } info ;
} ;
```

Common Trace Data

```
<<common trace fields>>=
MRT_EventType eventType ;
MRT_EventCode eventNumber ;
MRT_Instance *sourceInst ;
MRT_Instance *targetInst ;
```

`eventType`
: The type of the event that was dispatched.

`eventNumber`
: The number of the event that was dispatched.

sourceInst
A pointer to the instance that was the source of the dispatched event.

targetInst
A pointer to the instance that was the target of the dispatched event.

Transition Event Trace Data

transitionTrace

```
<<transition trace fields>>=
struct transitiontrace {
    MRT_StateCode currentState ;
    MRT_StateCode newState ;
} transitionTrace ;
```

currentState
The current state of the instance before the event dispatch.

newState
The new state entered as a result of the transition.

Polymorphic Event Trace Data

polyTrace

```
<<polymorphic trace fields>>=
struct polytrace {
    MRT_SubclassCode subcode ;
    MRT_DispatchCount genNumber ;
    MRT_EventCode mappedEvent ;
} polyTrace ;
```

subcode
The subclass code of the currently related instance.

genNumber
The number of the generalization down which the event was dispatched.

mappedNumber
The new event number to which the polymorphic event mapped.

The subclass type is encoded as a small integer value.

```
<<mrt internal trace simple types>>=
typedef uint8_t MRT_SubclassCode ;
```

Creation Event Trace Data

creationTrace

```
<<creation trace fields>>=
struct creationtrace {
    MRT_Class const *targetClass ;
} creationTrace ;
```

targetClass
A pointer to the class structure for the target of the creation event.

Access to Trace Information

Event tracing information is passed out of the run-time by having the application register a callback function. That function takes a pointer to the trace information as its argument.

```
<<mrt interface trace aggregate types>>=
typedef void (*MRT_TraceHandler)(MRT_TraceInfo const *) ;
```

The function is registered with the run-time by invoking:

```
<<mrt trace external interfaces>>=
extern MRT_TraceHandler
mrt_RegisterTraceHandler(
    MRT_TraceHandler handler) ;
```

handler
A pointer to a function that handles trace information.

The `mrt_RegisterTraceHandler` function is used to supply a trace callback function. Supplying a non-NULL value for *handler* also enables tracing. The return value of the the function is the previous value of the callback. Tracing can be turned off by invoking `mrt_RegisterTraceHandler` with `NULL`.

The trace handling function is invoked with a pointer to the trace information for the event being dispatched. The data pointed to by the argument is not valid after the return from the trace handler function. So handler functions must copy the trace data if they wish it to be available after their return.

The current value of the tracing function is stored in a static variable.

```
<<mrt trace static data>>=
static MRT_TraceHandler mrtTraceHandler ;
```

```
<<mrt trace external functions>>=
MRT_TraceHandler
mrt_RegisterTraceHandler(
    MRT_TraceHandler handler)
{
    MRT_TraceHandler oldhandler = mrtTraceHandler ;
    mrtTraceHandler = handler ;
    return oldhandler ;
}
```

The implementation of registering a handler is simply to record the function pointer in a variable.

For each type of event dispatch, the run-time calls a specific function to determine if tracing is enabled and to marshal the trace information into the proper data structure.

```
<<mrt trace static functions>>=
static inline void
mrtTraceTransitionEvent(
    MRT_EventCode event,
    MRT_Instance *source,
    MRT_Instance *target,
    MRT_StateCode currentState,
    MRT_StateCode newState)
{
    if (mrtTraceHandler) {
        MRT_TraceInfo trace = {
            .eventType = mrtTransitionEvent,
            .eventNumber = event,
```

```
            .sourceInst = source,
            .targetInst = target,
            .info.transitionTrace = {
                .currentState = currentState,
                .newState = newState
            }
        } ;
        mrtTraceHandler(&trace) ;
    }
}
```

```
<<mrt trace static functions>>=
static inline void
mrtTracePolymorphicEvent(
    MRT_EventCode event,
    MRT_Instance *source,
    MRT_Instance *target,
    MRT_SubclassCode subclass,
    MRT_DispatchCount genNumber,
    MRT_EventCode newEvent)
{
    if (mrtTraceHandler) {
        MRT_TraceInfo trace = {
            .eventType = mrtPolymorphicEvent,
            .eventNumber = event,
            .sourceInst = source,
            .targetInst = target,
            .info.polyTrace = {
                .subcode = subclass,
                .genNumber = genNumber,
                .mappedEvent = newEvent
            }
        } ;
        mrtTraceHandler(&trace) ;
    }
}
```

```
<<mrt trace static functions>>=
static inline void
mrtTraceCreationEvent(
    MRT_EventCode event,
    MRT_Instance *source,
    MRT_Instance *target,
    MRT_Class const *class)
{
    if (mrtTraceHandler) {
        MRT_TraceInfo trace = {
            .eventType = mrtCreationEvent,
            .eventNumber = event,
            .sourceInst = source,
            .targetInst = target,
            .info.creationTrace = {
                .targetClass = class
            }
        } ;
        mrtTraceHandler(&trace) ;
    }
}
```

Obtaining Tracing Output

A default trace handler is supplied that simply prints the trace information to the standard output using `printf`. Two version are supplied. One for when strings are present and the other for when they are not and only numbers can be printed.

```
<<mrt trace static functions>>=
#ifndef MRT_NO_STDIO
#ifndef MRT_NO_NAMES
static void
mrtPrintTraceInfo(
    MRT_TraceInfo const *traceInfo)
{
    char const *sourceName ;
    char const *sourceClassName ;
    char sourceIdNum[32] ;

    if (traceInfo->sourceInst == NULL) {
        sourceName = "?" ;
        sourceClassName = "?" ;
    } else {
        sourceClassName = traceInfo->sourceInst->classDesc->name ;
        sourceName = traceInfo->sourceInst->name ;
        if (sourceName == NULL) {
            unsigned instid = mrt_InstanceIndex(traceInfo->sourceInst) ;
            snprintf(sourceIdNum, sizeof(sourceIdNum), "%u", instid) ;
            sourceName = sourceIdNum ;
        }
    }

    char const *targetName = traceInfo->targetInst->name ;
    char targetIdNum[32] ;
    if (targetName == NULL) {
        unsigned instid = mrt_InstanceIndex(traceInfo->targetInst) ;
        snprintf(targetIdNum, sizeof(targetIdNum), "%u", instid) ;
        targetName = targetIdNum ;
    }

    switch (traceInfo->eventType) {
    case mrtTransitionEvent: {
        MRT_StateCode newState = traceInfo->info.transitionTrace.newState ;
        char const *newStateName ;
        if (newState == MRT_StateCode_IG) {
            newStateName = "IG" ;
        } else if (newState == MRT_StateCode_CH) {
            newStateName = "CH" ;
        } else {
            newStateName = traceInfo->targetInst->classDesc->edb->stateNames[
                traceInfo->info.transitionTrace.newState] ;
        }

        printf("%s: Transition: %s.%s - %s -> %s.%s: %s ==> %s\n",
            mrtTimestamp(), sourceClassName, sourceName,
            traceInfo->targetInst->classDesc->eventNames[traceInfo->eventNumber],
            traceInfo->targetInst->classDesc->name, targetName,
            traceInfo->targetInst->classDesc->edb->stateNames[
                traceInfo->info.transitionTrace.currentState],
            newStateName) ;
    }
        break ;

    case mrtPolymorphicEvent: {
        MRT_Relationship const *rel = traceInfo->targetInst->classDesc->pdb->
```

```
                genDispatch[traceInfo->info.polyTrace.genNumber].relship ;
        MRT_Class const *subclass ;
        char const *subname = NULL ;
        if (rel->relType == mrtRefGeneralization) {
            subclass = rel->relInfo.refGeneralization.
                subclasses[traceInfo->info.polyTrace.subcode].classDesc ;
            subname = subclass->name ;
        } else if (rel->relType == mrtUnionGeneralization) {
            subclass = rel->relInfo.unionGeneralization.
                subclasses[traceInfo->info.polyTrace.subcode] ;
            subname = subclass->name ;
        } else {
            printf("%s: bad relationship type in polymorphic event, %d\n",
                mrtTimestamp(), rel->relType) ;
            break ;
        }
        printf("%s: Polymorphic: %s.%s - %s -> %s.%s: %s - %s -> %s\n",
            mrtTimestamp(), sourceClassName, sourceName,
            traceInfo->targetInst->classDesc->eventNames[traceInfo->eventNumber],
            traceInfo->targetInst->classDesc->name, targetName,
            traceInfo->targetInst->classDesc->pdb->genNames[
                traceInfo->info.polyTrace.genNumber],
            subclass->eventNames[traceInfo->info.polyTrace.mappedEvent],
            subname) ;
    }
        break ;

    case mrtCreationEvent:
        printf("%s: Creation: %s.%s - %s -> %s ==> %s\n",
            mrtTimestamp(), sourceClassName, sourceName,
            traceInfo->targetInst->classDesc->eventNames[traceInfo->eventNumber],
            traceInfo->info.creationTrace.targetClass->name,
            targetName) ;
        break ;

    default:
        printf("%s: Unknown trace event type, \"%u\"",
                mrtTimestamp(), traceInfo->eventType) ;
        break ;
    }
}
#   else  /* MRT_NO_NAMES is defined */
static void
mrtPrintTraceInfo(
    MRT_TraceInfo const *traceInfo)
{
    switch (traceInfo->eventType) {
    case mrtTransitionEvent:
        printf("%s: Transition: %p - %u -> %p: %u ==> %u\n",
                mrtTimestamp(), traceInfo->sourceInst, traceInfo->eventNumber,
                traceInfo->targetInst,
                traceInfo->info.transitionTrace.currentState,
                traceInfo->info.transitionTrace.newState) ;
        break ;

    case mrtPolymorphicEvent:
        printf("%s: Polymorphic: %p - %u -> %p: %u - %u -> %d\n",
                mrtTimestamp(), traceInfo->sourceInst, traceInfo->eventNumber,
                traceInfo->targetInst,traceInfo->info.polyTrace.genNumber,
                traceInfo->info.polyTrace.mappedEvent,
                traceInfo->info.polyTrace.subcode) ;
        break ;
```

```
    case mrtCreationEvent:
        printf("%s: Creation: %p - %u -> %p ==> %p\n",
                mrtTimestamp(), traceInfo->sourceInst, traceInfo->eventNumber,
                traceInfo->info.creationTrace.targetClass,
                traceInfo->targetInst) ;
        break ;

    default:
        printf("%s: Unknown trace event type, \"%u\"",
                mrtTimestamp(), traceInfo->eventType) ;
        break ;
    }
}
#endif /* MRT_NO_NAMES */
#endif /* MRT_NO_STDIO */
```

```
<<mrt forward references>>=
#ifndef MRT_NO_TRACE
static char const *mrtTimestamp(void) ;
#endif /* MRT_NO_TRACE */
```

```
<<mrt trace external functions>>=
#ifndef MRT_NO_TRACE
static char const *
mrtTimestamp(void)
{
    static char ts_buf[32] ;
    (void)sys_util_eptime_ticks_format(NULL, sizeof(ts_buf), ts_buf) ;
    return ts_buf ;
}
#endif /* MRT_NO_TRACE */
```

Tracing Strategies

Clearly, tracing can generate data at a rather high rate and can be rather intrusive upon the execution of the system. Several strategies may be used to deal with the trace data. If possible, all the trace data can be dumped in a raw form out a communications interface and let some other program decode and display it. That may still be too intrusive and sometimes it is best to filter the trace data based on the target instance pointer value. In this way you may trace the event dispatches on only a subset of instances. Several different filtering schemes, such as source instance or classes, can be envisioned.

Another possibility is to store trace information in a memory area in some sort of circular queue arrangement. Then it is possible for the application to start and stop such tracing and achieve “logic analyzer” type triggering functionality. The trace information can then be extracted from memory and analyzed.

When running in a POSIX environment, one can assume reasonable I/O facilities. The POSIX version of the run-time includes default trace handling to timestamp and print the trace data in human readable form.

You will also note that the trace information has no timing data associated with it. This type of data is so system specific that it is left to the tracing callback to supply. If you have a free running cycle counter in your system, this can be a good indicator of relative time and the trace callback function can add this to the data set supplied by the run-time. Your system may also have source of clocked timing data that can also be used as a timing reference. In either case, augmenting the trace data with some sort of relative time information is very valuable.

Tracing can also be used as a framework for testing. If a domain is built to run in a testing framework where tracing is enabled, then recording all the trace information allows one to determine the amount of *transition* coverage a test set causes. The goal is to develop test sets that drive the domain with appropriate data so that all state transitions are taken. Tracing allows the recording of what transitions a given thread of control causes. Since in most well designed state machines, state activity code is small and does not contain complicated or intricate internal program flow, causing all state activities to be executed is often close to

complete statement coverage. As an added benefit, state machines can be considered as directed graphs. A depth first traversal of a directed graph can be be used to determine a spanning tree for the graph. Traversing a spanning tree for a graph insures that all nodes in the graph are visited and the event sequence given by the spanning tree can guide the generation of test set data and can help to minimize the number of test cases required to ensure adequate coverage.

Error Handling

Until now we have glossed over the subject of how to handle errors in the `micca` run-time. In XUML, the domains assume a perfect architecture in the sense that no formal mechanism is provided to signal architectural errors back to the application domains. This makes sense because the application models are meant to be implementation independent and able to be run on a variety of underlying platforms. However, an error policy, in much the same terms as data and execution policies, must be put into place. The details of the error handling policy will vary between software architectures, so it is important to state them clearly. For the `micca` run-time, the following principles guide error handling.

- To that extent possible, the run-time operations should not report errors back to the application. For implementation languages that do not support exception handling, the usual technique of returning error codes is not very effective. Either by accident or sloth, many error codes are not checked. Even when the error code is checked, there is little recovery recourse for the application. For example, it does little good to know that we are unable to generate an event because we do not have sufficient ECB resources when there is nothing a state activity can do to free up the required resources.
- Errors that result from exhausted resources or analysis errors detected at run-time are **fatal**. Exactly how fatal errors are acted upon is platform dependent and may result in terminating a program or completely resetting the system. Regardless of the consequence of a fatal error, the assumption is that the program can no longer continue to run.

With these principles in mind, we define a set of error conditions that are detected by the run-time. All these conditions are fatal and are handled by invoking a fatal error handler.

```
<<mrt interface simple types>>=
typedef enum {
    mrtCantHappen = 1,
    mrtEventInFlight,
    mrtNoECB,
    mrtNoInstSlot,
    mrtUnallocSlot,
    mrtSyncOverflow,
    mrtRefIntegrity,
    mrtTransOverflow,
    mrtInstSetOverflow,
    mrtStaticRelationship,
    mrtRelationshipLinkage,
    mrtDupAssociator,
    mrtPanic,
    mrtTimerOpFailed,

#       ifdef _POSIX_C_SOURCE
    mrtSignalOpFailed,
    mrtSelectWaitFailed,
#       endif /* _POSIX_C_SOURCE */
} MRT_ErrorCode ;
```

We will need a string representation of the error codes to make human readable messages.

```
<<mrt static data>>=
static char const * const mrtErrorMsgs[] = {
    [0] = "no error",     /* place holder */
    [mrtNoECB] = "no available Event Control Blocks\n",
    [mrtSyncOverflow] = "synchronization queue overflow\n",
    [mrtTransOverflow] = "transaction markings overflow\n",
```

```
    [mrtInstSetOverflow] = "instance set overflow: instance number %u\n",
    [mrtStaticRelationship] = "attempt to modify static relationship\n",
    [mrtRelationshipLinkage] = "invalid instance linkage operation or value\n",
    [mrtPanic] = "panic: %s\n",
    [mrtTimerOpFailed] = "timer operation failed: %s\n",

#       ifndef MRT_NO_NAMES
    [mrtCantHappen] = "can't happen transition: %s.%s: %s - %s -> CH\n",
    [mrtEventInFlight] = "event-in-flight error: %s.%s - %s -> %s.%s\n",
    [mrtNoInstSlot] = "no available instance slots: %s\n",
    [mrtUnallocSlot] = "unallocated instance slot: %u in class %s\n",
    [mrtRefIntegrity] = "referential integrity check failed: %s\n",
    [mrtDupAssociator] = "duplicate associator instance: %s\n",
#       else
    [mrtCantHappen] = "can't happen transition: %p: %u - %u -> CH\n",
    [mrtEventInFlight] = "event-in-flight error: %p - %u -> %p\n",
    [mrtNoInstSlot] = "no available instance slots: %p\n",
    [mrtUnallocSlot] = "unallocated instance slot: %u in class %p\n",
    [mrtRefIntegrity] = "referential integrity check failed: %p\n",
    [mrtDupAssociator] = "duplicate associator instance: %p\n",
#       endif /* MRT_NO_NAMES */

#       ifdef _POSIX_C_SOURCE
    [mrtSignalOpFailed] = "signal operation failed: %s\n",
    [mrtSelectWaitFailed] = "blocking on pselect() failed: %s\n",
#       endif /* _POSIX_C_SOURCE */
} ;
```

For some of the error messages, there are parameters which containing naming information. Above we defined two versions of the format string for the error message. When names are not present, the best we can usually do is to print pointer addresses. Below we define inline functions to handle the difference when naming information is included and when it is not. This allows us to have a single interface and avoid sprinkling conditional compilation directives through the code.

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtCantHappenError(
    MRT_Instance *const targetInst,
    MRT_StateCode currentState,
    MRT_EventCode eventNumber)
{
    mrtFatalError(mrtCantHappen,
        targetInst->classDesc->name,
        targetInst->name ? targetInst->name : "?",
        targetInst->classDesc->edb->stateNames[currentState],
        targetInst->classDesc->eventNames[eventNumber]) ;
}

#else

noreturn static void inline
mrtCantHappenError(
    MRT_Instance *const targetInst,
    MRT_StateCode currentState,
    MRT_EventCode eventNumber)
{
    mrtFatalError(mrtCantHappen, targetInst, currentState, eventNumber) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES
noreturn static void inline
mrtEventInFlightError(
    MRT_Instance *const sourceInst,
    MRT_EventCode eventNumber,
    MRT_Instance *const targetInst)
{
    mrtFatalError(mrtEventInFlight,
            sourceInst ? sourceInst->classDesc->name : "?",
            sourceInst ? sourceInst->name : "?",
            targetInst->classDesc->eventNames[eventNumber],
            targetInst->classDesc->name,
            targetInst->name ? targetInst->name : "?") ;
}

#else

noreturn static void inline
mrtEventInFlightError(
    MRT_Instance *const sourceInst,
    MRT_EventCode eventNumber,
    MRT_Instance *const targetInst)
{
    mrtFatalError(mrtEventInFlight, sourceInst, eventNumber, targetInst) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtNoInstSlotError(
    MRT_Class const *const classDesc)
{
    mrtFatalError(mrtNoInstSlot, classDesc->name) ;
}

#else

noreturn static void inline
mrtNoInstSlotError(
    MRT_Class const *const classDesc)
{
    mrtFatalError(mrtNoInstSlot, classDesc) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtUnallocSlotError(
    MRT_InstId slot,
    MRT_Class const *const classDesc)
{
    mrtFatalError(mrtUnallocSlot, slot, classDesc->name) ;
}
```

```
#else

noreturn static void inline
mrtUnallocSlotError(
    MRT_InstId slot,
    MRT_Class const *const classDesc)
{
    mrtFatalError(mrtUnallocSlot, slot, classDesc) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtRefIntegrityError(
    MRT_Relationship const *const rel)
{
    mrtFatalError(mrtRefIntegrity, rel->name) ;
}

#else

noreturn static void inline
mrtRefIntegrityError(
    MRT_Relationship const *const rel)
{
    mrtFatalError(mrtRefIntegrity, rel) ;
}

#endif /* MRT_NO_NAMES */
```

```
<<mrt implementation static inlines>>=
#ifndef MRT_NO_NAMES

noreturn static void inline
mrtDupAssociatorError(
    MRT_Relationship const *const rel)
{
    mrtFatalError(mrtDupAssociator, rel->name) ;
}

#else

noreturn static void inline
mrtDupAssociatorError(
    MRT_Relationship const *const rel)
{
    mrtFatalError(mrtDupAssociator, rel) ;
}

#endif /* MRT_NO_NAMES */
```

Everywhere else the run-time operations have been crafted to avoid error possibilities. For example, as discussed in delayed event generation, we interpret the attempt to generate a duplicate delayed event as wishing to cancel the existing one and instantiate a new one. This semantic interpretation avoids generating an error and avoids all the additional code require in state activities that generate delayed events.

Exactly how fatal errors are handled will depend upon the specifics of how the platform handles errors. We define an interface for a fatal error handler.

```
<<mrt interface aggregate types>>=
typedef void (*MRT_FatalErrorHandler)(MRT_ErrorCode, char const *, va_list) ;
```

The interface is patterned after `vprintf`, giving a format string and a pointer to a variable length argument list.

```
<<mrt forward references>>=
static void
mrtDefaultFatalErrorHandler(
    MRT_ErrorCode errNum,
    char const *fmt,
    va_list ap) ;
```

The system provides a default fatal error handler, a message is printed to the standard error stream.

```
<<mrt static functions>>=
static void
mrtDefaultFatalErrorHandler(
    MRT_ErrorCode errNum,
    char const *fmt,
    va_list ap)
{
#       ifndef MRT_NO_STDIO
    vfprintf(stderr, fmt, ap) ;
#       endif /* MRT_NO_STDIO */
}
```

A pointer to the fatal error handler is initialized with the default one.

```
<<mrt static data>>=
static MRT_FatalErrorHandler mrtErrHandler = mrtDefaultFatalErrorHandler ;
```

Because fatal error handling is usually so platform specific and because of the need to test fatal error paths, we provide the ability to delegate the consequence of the fatal error.

```
<<mrt external interfaces>>=
extern MRT_FatalErrorHandler
mrt_SetFatalErrorHandler(
    MRT_FatalErrorHandler newHandler) ;
```

`newHandler`
A pointer to a fatal error handler function.

The `MRT_FatalErrorHandler` function install `newHandler` as the fatal error handler and return the previous error handler function pointer.

```
<<mrt external functions>>=
MRT_FatalErrorHandler
mrt_SetFatalErrorHandler(
    MRT_FatalErrorHandler newHandler)
{
    MRT_FatalErrorHandler prevHandler = mrtErrHandler ;
    if (newHandler) {
        mrtErrHandler = newHandler ;
    }
    return prevHandler ;
}
```

The run-time internally calls `mrtFatalError`.

```
<<mrt forward references>>=
noreturn static void
mrtFatalError(MRT_ErrorCode errNum, ...) ;
```

```
<<mrt static functions>>=
noreturn static void
mrtFatalError(
    MRT_ErrorCode errNum,
    ...)
{
    va_list ap ;

    assert(mrtErrHandler != NULL) ;
    assert(errNum < COUNTOF(mrtErrorMsgs)) ;

    va_start(ap, errNum) ;
    mrtErrHandler(errNum, mrtErrorMsgs[errNum], ap) ;
    va_end(ap) ;
    /*
     *  If the handler does return, we insist that all errors
     *  are fatal. So we abort().
     */
    abort() ;
}
```

As we see in the code, we insist that there is an error handler. There is always the default one and a different handler can be specified if necessary. Finally, `abort()` is called should the error handler return.

Avoiding Fatalities

In this error handling strategy, every run-time detected error is fatal. Although the details of the processing for fatal errors can be delegated, in most systems of the class we consider here, fatal errors usually result in a system reset. Under the vast majority of circumstances, that is the desired behavior. However, there are some particular circumstances where causing a fatal error is not the desired behavior.

Consider the case where some external stimulus results in an event being generated. If the stimulus occurs more frequently than events can be processed, then the run-time will run out of event control blocks causing a fatal error. As an example, consider the arrival of a communications packet. If somewhere during the processing of the packet an event is generated, then if packets arrive too fast a fatal error can be generated. In effect it would provide a means for an external stimulus to cause the system to crash. Certainly for the case of a communications packet, the preferred behavior would be to drop the packet and let higher level protocol deal with the necessary retries, etc. It is then necessary to be able to determine if generating an event would be successful.

In this section we describe functions that can be used to avoid run-time requests that would exhaust an underlying resource and therefore cause a fatal system error. It should be emphasized that these functions are **not** intended for ordinary or casual use. Under the vast majority of circumstances, such as when one state machine generates an event to another state machine, event generation and other such activities should continue to assume that there are no resources that can be consumed. System analysis and testing should then determine the appropriate sizing for the various resource pools. The capability described in this section is to handle unusual and extraordinary circumstances where hardware failure or failure to abide by communications protocols could force the system into a fatal error situation.

Note also that the alternative provided here causes the external stimulus that would cause the fatal error condition effectively to be ignored. For some system requirements that is not an acceptable solution. For example, consider a digital input line that is used to generate an interrupt and that interrupt signals an external condition monitored by hardware, say the maximum extent of motion of physical robot arm. If this interrupt arrives at a very fast rate, one might conclude the hardware has failed. Ignoring the interrupt might do little other than mask a problem that should cause a fatal error condition and potentially reset the system. The conclusion is that providing a means of avoiding fatal error conditions is not intended to serve as an overall error handling policy. Careful analysis and consideration is still required. If an interrupt arrives faster than expected and, because of what the

interrupt represents, it cannot be ignored, that fact and the response to it must be deduced by the interrupt service code (*e.g.* by determining the interrupt frequency) and actions appropriate to the system must be taken. It can be a difficult problem to solve and the functions provided here are too generic to be used indiscriminately.

There are three internal run-time resources that can be exhausted.

- Class instances can be dynamically created and each class has its own instance pool.
- Event control blocks are used for generating and dispatching state machine events.
- The foreground / background synchronization queue has a fixed number of slots and excessive synchronization requests from interrupt service routines can fill the queue.

Checking for Available Instance Space

The `mrt_CanCreateInstance()` function provides a means determine if there is space available to create a new instance of a class.

```
<<mrt internal external interfaces>>=
extern bool
mrt_CanCreateInstance(
    MRT_Class const *const classDesc) ;
```

`classDesc`
A pointer to the class data for which the space check is made.

`mrt_CanCreateInstance` returns `true` if there at least one instance of `classDesc` may be created without exhausting the memory pool of instances for the class.

```
<<mrt external functions>>=
bool
mrt_CanCreateInstance(
    MRT_Class const *const classDesc)
{
    assert(classDesc != NULL) ;

    /*
     * Search for an empty slot in the pool.
     */
    return mrtFindInstSlot(classDesc->iab) != NULL ;
}
```

Checking for Event Blocks

```
<<mrt external interfaces>>=
extern bool
mrt_CanSignalEvent(void) ;
```

The `mrt_CanSignalEvent` function returns true if it is possible to signal an event and *not* cause a system error.

```
<<mrt external functions>>=
bool
mrt_CanSignalEvent(void)
{
    return !mrtEventQueueEmpty(&freeEventQueue) ;
}
```

Causing Fatalities

There are rare cases when executing outside of the state machine dispatch mechanism where situations are detected where the execution state cannot be resolved and it is necessary to assert a *panic* situation. For this situation, the run time provides the `mrt_Panic` function.

```
<<mrt external interfaces>>=
extern noreturn void
mrt_Panic(
    char const *format,
    ...) ;
```

`format`
: A `printf` style format string.

`...`
: A variable length list of arguments matching the `format` string conversion specifiers.

The `mrt_Panic` formats an output string according to the `format` argument and the other invocation parameters and forces an error situation.

```
<<mrt external functions>>=
noreturn void
mrt_Panic(
    char const *format,
    ...)
{
    char outbuf[128] ;
    va_list ap ;

    va_start(ap, format) ;
    vsnprintf(outbuf, sizeof(outbuf), format, ap) ;
    va_end(ap) ;

    mrtFatalError(mrtPanic, outbuf) ;
}
```

Linked List Operations

The run-time supplies a set of linked list operations for use in manipulating the lists of instance pointers used to realize and navigate associations. They are the usual circular, doubly linked list manipulations and are presented here without little further comment.

```
<<mrt internal static inlines>>=
static inline MRT_LinkRef *
mrtLinkRefBegin(
    MRT_LinkRef const *list)
{
    return list->next ;
}
```

```
<<mrt internal static inlines>>=
static inline MRT_LinkRef *
mrtLinkRefEnd(
    MRT_LinkRef const *list)
{
```

```
    return (MRT_LinkRef *)list ;
}
```

```
<<mrt internal static inlines>>=
static inline bool
mrtLinkRefEmpty(
    MRT_LinkRef const *list)
{
    return list->next == list ;
}
```

```
<<mrt internal static inlines>>=
static inline bool
mrtLinkRefNotEmpty(
    MRT_LinkRef const *list)
{
    return list->next != list ;
}
```

Note that an empty list is one where the list terminus points back to itself.

```
<<mrt implementation static inlines>>=
static inline void
mrtLinkRefInit(
    MRT_LinkRef *ref)
{
    ref->next = ref->prev = ref ;
}
```

Because the list is doubly linked, we only have to have a pointer to the item to remove it.

```
<<mrt implementation static inlines>>=
static inline void
mrtLinkRefRemove(
    MRT_LinkRef *item)
{
    item->prev->next = item->next ;
    item->next->prev = item->prev ;
    item->next = item->prev = NULL ; // ❶
}
```

❶ We set the values of the `next` and `prev` members to `NULL` when the item has been removed from a list. This gives a handy test to determine if the item is in any list. This is useful in some of the manipulations of linked lists of references in order to maintain such lists as a set.

The insert function inserts an item before a given point in the list. Since it is usually called with a list head as the insertion place, the net effect is to place the item last in the list.

```
<<mrt implementation static inlines>>=
static inline void
mrtLinkRefInsert(
    MRT_LinkRef *item,
    MRT_LinkRef *at)
{
    if (item->next != NULL || item->prev != NULL) { // ❶
        mrtFatalError(mrtRelationshipLinkage) ;
    }
    item->prev = at->prev ;
    item->next = at ;
```

```
    at->prev->next = item ;
    at->prev = item ;
}
```

(1) We also test that they are `NULL` before inserting an item into a list to insure that we are not inserting a duplicate. Inserting an item that is already in another list destroys the pointer structure of the list.

Code Layout

```
<<micca_rt.h>>=
<<edit warning>>
<<copyright info>>

/*
Micca version:
<<version info>>
*/

#ifndef MICCA_RT_H_
#define MICCA_RT_H_

/*
 * Includes
 */
<<mrt interface includes>>

/*
 * Constants
 */
<<mrt interface constants>>

/*
 * Data Types
 */
<<mrt interface simple types>>
<<mrt interface aggregate types>>

#   ifndef MRT_NO_TRACE
<<mrt interface trace aggregate types>>
#   endif /* MRT_NO_TRACE */

/*
 * External Functions
 */
<<mrt external interfaces>>

#   ifndef MRT_NO_TRACE
<<mrt trace external interfaces>>
#   endif /* MRT_NO_TRACE */

#endif /* MICCA_RT_H_ */
```

```
<<micca_rt.c>>=
<<edit warning>>
<<copyright info>>

/*
Micca version:
<<version info>>
```

```
*/

/*
 * Includes
 */
#include "micca_rt.h"
#include "micca_rt_internal.h"
#include "useful.h"
<<mrt implementation includes files>>

/*
 * Constants
 */
<<mrt implementation constants>>

/*
 * Data Types
 */
<<mrt implementation simple types>>
<<mrt implementation aggregate types>>

/*
 * Forward References
 */
<<mrt forward references>>

/*
 * Static Data
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static data>>
#   endif /* MRT_NO_TRACE */

<<mrt static data>>

/*
 * Static Inline Functions
 */
<<mrt implementation static inlines>>

/*
 * Static Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace static functions>>
#   endif /* MRT_NO_TRACE */

<<mrt static functions>>

/*
 * External Functions
 */
#   ifndef MRT_NO_TRACE
<<mrt trace external functions>>
#   endif /* MRT_NO_TRACE */

<<mrt external functions>>
```

Internal Header File

The run-time code uses an internal header file to separate declarations that should not be generally exposed. This header file, called `micca_rt_internal.h`, must be included in the generated domain code, but contains definitions that are not needed by bridge code.

```
<<mrt interface includes>>=
#include <stddef.h>
#include <stdbool.h>
#include <inttypes.h>
#include <stdarg.h>
#include <stdnoreturn.h>
#include <assert.h>
```

```
<<micca_rt_internal.h>>=
<<edit warning>>
<<copyright info>>

/*
Micca version:
<<version info>>
*/

#ifndef MICCA_RT_INTERNAL_H_
#define MICCA_RT_INTERNAL_H_

/*
 * Standard Includes
 */
#ifndef MRT_NO_STDIO
#   include <stdio.h>
#endif /* MRT_NO_STDIO */

#include <stdlib.h>
#include <string.h>

#if __STDC_VERSION__ >= 201112L         /* ❶ */
#   include <stdalign.h>
#   include <stdnoreturn.h>
#else
#   ifndef noreturn
#       ifdef __CC_ARM
#           define noreturn __declspec(noreturn)
#       else /* __CC_ARM */
#           define noreturn
#       endif /* __CC_ARM */
#   endif /* noreturn */
#   ifndef alignas
#       define alignas(x)
#   endif /* alignas */
#endif /* __STDC_VERSION__ >= 201112L */

/*
 * Includes
 */
<<mrt internal includes>>

/*
 * Constants
 */
<<mrt internal constants>>
```

```
/*
 * Data Types
 */
<<mrt internal simple types>>
<<mrt internal aggregate types>>
#   ifndef MRT_NO_TRACE
<<mrt internal trace simple types>>
<<mrt internal trace aggregate types>>
#   endif /* MRT_NO_TRACE */

/*
 * External Functions
 */
<<mrt internal external interfaces>>

/*
 * Static Inline Functions
 */
<<mrt internal static inlines>>

#endif /* MICCA_RT_INTERNAL_H_ */
```

1. Although we target the C11 standard syntax, there are really only a few C11 features that we use. So, for those stuck with a C99 compiler, we can use the preprocessor the remove the C11 dependencies. The most serious implication is over the `alignas` macro. We seek to have the most liberal alignment we can obtain, and if that is not the default by a C99 compiler, then the declaration of event parameters and sync function parameters could cause problems.

Part III

Canopy

Part III of the book is not in place. It will discuss the modeling language and conventions which are used to scale software design solutions that directly translate onto the software mechanisms provided in Part II.

Chapter 12

Life at the Top of the Forest

Introduction

Part IV

Supporting code

Chapter 13

Bit Manipulation

When dealing with bit-encoded hardware registers, you need a set of functions to handle the low level bit encodings. There are many ways to accomplish bit manipulations. Using `static inline` functions are preferred in this implementation. Also notice that "C" structure bit fields are avoided. Bit field order in "C" structures is dependent upon the compiler implementation.

Defining bit fields

Often bit fields are defined in terms of:

1. the offset from bit 0 to the beginning of the field and
2. the number of contiguous bits in the field.

CMSIS header files take a different approach to bit field manipulations. For CMSIS, bit fields are defined by:

1. the offset from bit 0 to the beginning of the field and
2. a mask containing a contiguous sequence of 1-bits defining the length of the field.

Both approaches are used as necessary.

Bit masks

btwd_mask

Generate a bit mask with `n` contiguous 1 bits starting at `o` bit position.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_mask(
    uint32_t n,
    uint32_t o)
{
    return ((1 << (n)) - 1) << o ;
}
```

n	the number of contiguous bits in the bit field.
o	the offset of the bit field within a word.

btwd_field_max

Generate the maximum unsigned number which can be contained in a bit field of a given width.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_field_max(
    uint32_t n)
{
    return ((1 << (n)) - 1) ;
}
```

n	the number of bits in the bit field.

btwd_bit_mask

Generate a single bit mask. This is common enough to deserve its own function.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_bit_mask(
    uint32_t o)
{
    return btwd_mask(1, o) ;
}
```

o	the offset of the bit field within a word.

btwd_bit_clear

Clear a single bit in `w` at offset, `o`.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_bit_clear(
    uint32_t w,
    uint32_t o)
{
    return w & ~btwd_bit_mask(o) ;
}
```

w	the word value to be modified.
o	the offset of the bit field within a word.

btwd_bit_set

Set a single bit in `w` at offset, `o`.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_bit_set(
    uint32_t w,
    uint32_t o)
{
    return w | btwd_bit_mask(o) ;
}
```

w	the word value to be modified.
o	the offset of the bit field within a word.

btwd_bit_toggle

Toggle a single bit in `w` at offset, `o`.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_bit_toggle(
    uint32_t w,
    uint32_t o)
{
    return w ^ btwd_bit_mask(o) ;
}
```

w	the word value to be modified.
o	the offset of the bit field within a word.

btwd_bit_test

Test if a bit is set.

```
<<bit_twiddle.h: inline functions>>=
static inline bool
btwd_bit_test(
    uint32_t w,
    uint32_t o)
{
    return (w & btwd_bit_mask(o)) != 0 ;
}
```

w	the word value to be modified.
o	the offset of the bit field within a word.

btwd_bits_insert

Insert bits into a bit field.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_bits_insert(
    uint32_t r,
    uint32_t v,
    uint32_t w,
    uint32_t o)
{
    uint32_t mask = btwd_mask(w, o) ;
    return (r & ~mask) | ((v << o) & mask) ;
}
```

r	the value into which bits are inserted
v	the bits to insert
w	the width of the bit field
o	the offset of the bit field within a word.

btwd_bits_extract

Extract bits from a bit field.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_bits_extract(
    uint32_t r,
    uint32_t w,
    uint32_t o)
{
    uint32_t mask = btwd_mask(w, o) ;
    return (r & mask) >> o ;
}
```

r	the value from which bits are extracted
w	the width of the bit field
o	the offset of the bit field within a word.

CMSIS style field operations

When an ARM SVD file is processed by the `SVDConf` program, one program option is to emit into a generated header file a set of pre-processor macros that define all the register bit fields in the SVD file[1]. The functions in this section are designed to handle the generated CMSIS macros. Each bit field generates a `_Msk` symbol and a `_Pos` symbol. The `_Msk` symbol is a bit mask, *i.e.* a value where there are 1-bits in the bit positions of the field. The 1-bits are contiguous for most bit masks. The `_Pos` symbol is the offset to the first bit in the field counting from bit 0.

[1] There are other options available for how this information is generated

btwd_field_insert

Insert into `r` the value `v` that is masked by `m` and shifted into position by `o` bits. Here, `v` is shifted to the correct position in the bit field and OR'ed into the cleared bit field slot. *N.B.* that `v` is truncated to fit the bit field definition.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_field_insert(
    uint32_t r,
    uint32_t v,
    uint32_t m,
    uint32_t o)
{
    return (r & ~m) | ((v << o) & m) ;
}
```

r	the word value to be modified.
v	the field value.
m	the field mask.
o	the offset of the bit field within a word.

btwd_field_extract

Extract the field value in `r` using mask `m` that is at offset `o`.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_field_extract(
    uint32_t r,
    uint32_t m,
    uint32_t o)
{
    return (r & m) >> o ;
}
```

r	the word value to be modified.
m	the field mask.
o	the offset of the bit field within a word.

btwd_field_clear

Set to zero the bits in `r` whose mask is `m`.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_field_clear(
    uint32_t r,
    uint32_t m)
{
    return r & ~m ;
}
```

r	the word value to be modified.
m	the field mask.

btwd_field_set

Set to 1 the bits in `r` whose mask is `m`.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_field_set(
    uint32_t r,
    uint32_t m)
{
    return r | m ;
}
```

r	the word value to be modified.
m	the field mask.

btwd_field_test

Test the field in `r` whose mask is `m` to determine if any bits are set.

```
<<bit_twiddle.h: inline functions>>=
static inline bool
btwd_field_test(
    uint32_t r,
    uint32_t m)
{
    return (r & m) != 0 ;
}
```

r	the word value to be modified.
m	the field mask.

Accessing hardware registers

```
<<bit_twiddle.h: data type declarations>>=
typedef uint32_t volatile HDWR_Register ;
```

btwd_write_reg_field

Write the field value `v` into the register pointed to by `reg` with the field defined by its mask `m` and offset `o`. Returns the value written to the register.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_write_reg_field(
    HDWR_Register *const reg,
    uint32_t v,
    uint32_t m,
    uint32_t o)
{
    uint32_t r = *reg ;
    r = btwd_field_insert(r, v, m, o) ;
    *reg = r ;
    return r ;
}
```

reg	a pointer to a memory locations which holds a hardware register.
v	the field value.
m	the field mask.
o	the offset of the bit field within a word.

btwd_set_reg_field

Set to 1 all of the bits in the register pointed to by `reg` with the field defined by its mask `m`. Returns the value written to the register.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_set_reg_field(
    HDWR_Register *const reg,
    uint32_t m)
{
    return *reg |= m ;
}
```

reg	a pointer to a memory locations which holds a hardware register.
m	the field mask.

btwd_clear_reg_field

Set to 0 all of the bits in the register pointed to by `reg` with the field defined by its mask `m`. Returns the value written to the register.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_clear_reg_field(
    HDWR_Register *const reg,
    uint32_t m)
{
    return *reg &= ~m ;
}
```

reg	a pointer to a memory locations which holds a hardware register.
m	the field mask.

btwd_read_reg_field

Read the register pointed to by `reg` and return the field value which has mask `m` at offset `o`.

```
<<bit_twiddle.h: inline functions>>=
static inline uint32_t
btwd_read_reg_field(
    HDWR_Register *const reg,
    uint32_t m,
    uint32_t o)
{
    return btwd_field_extract(*reg, m, o) ;
}
```

reg	a pointer to a memory locations which holds a hardware register.
m	the field mask.
o	the offset of the bit field within a word.

btwd_test_reg_field

Test if the register pointed to by `reg` has a non-zero value in the bit field which has mask `m`.

```
<<bit_twiddle.h: inline functions>>=
static inline bool
btwd_test_reg_field(
    HDWR_Register *const reg,
    uint32_t m)
{
    return (*reg & m) != 0 ;
}
```

reg	a pointer to a memory locations which holds a hardware register.
m	the field mask.

Function Macros Using CMSIS Definitions

Generally, pre-processor macro functions are avoided. They are fraught with potential problems and they inhibit your knowing exactly what is being presented to the compiler. A static inline function is usually a better choice. However, since the CMSIS pre-processor symbols for register fields were defined with a naming convention, the pre-processor can be enlisted to save some typing by using the string concatenation operator.

The following macros are useful in conjunction with CMSIS style macros for hardware fields. These macros use a bit position and bit masks to control the values placed in an I/O register. CMSIS uses a naming convention of appending `_Pos` to a field name to indicate the bit offset and `_Msk` for the field mask. This leads naturally to the use of the "C" pre-processor to save some typing. Here are some pre-processor macro functions to insert and extract bit fields from a register knowing only the field name. The macros use the pre-processor capability to concatenate strings.

```
<<bit_twiddle.h: macros>>=
#define BTWD_FIELD_INSERT(r, v, f)  btwd_field_insert((r), (v), f ## _Msk, f ## _Pos)
#define BTWD_FIELD_EXTRACT(r, f)    btwd_field_extract((r), f ## _Msk, f ## _Pos)
#define BTWD_FIELD_CLEAR(r, f)      btwd_field_clear((r), f ## _Msk)
#define BTWD_FIELD_SET(r, f)        btwd_field_set((r), f ## _Msk)
#define BTWD_FIELD_TEST(r, f)       btwd_field_test((r), f ## _Msk)
#define BTWD_WRITE_REG_FIELD(reg, v, f)\
        btwd_write_reg_field((reg), (v), f ## _Msk, f ## _Pos)
#define BTWD_SET_REG_FIELD(reg, f)  btwd_set_reg_field((reg), f ## _Msk)
#define BTWD_CLEAR_REG_FIELD(reg, f) btwd_clear_reg_field((reg), f ## _Msk)
#define BTWD_READ_REG_FIELD(reg, f) btwd_read_reg_field((reg), f ## _Msk, f ## _Pos)
#define BTWD_TEST_REG_FIELD(reg, f) btwd_test_reg_field((reg), f ## _Msk)
```

btwd_align

Compute the ceiling of a number for memory address alignment purposes. Alignment is the exponent of the power of 2 for which the ceiling is calculated.

```
<<bit_twiddle.h: inline functions>>=
static inline size_t
btwd_align(
    size_t number,
    unsigned align_exp)
{
    assert(align_exp < 32) ;

    size_t const alignment = (1 << align_exp) - 1U ;
    return (number + alignment) & ~alignment ;
}
```

number	a value to be aligned
align_exp	the exponent of a power of 2 where number is aligned, *i.e.* 1 aligns to 2 bytes, 2 aligns to 4 bytes, etc.

Code Layout

Bit Twiddle Header File Layout

Bit twiddle header file

```
<<bit_twiddle.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   bit_twiddle.h -- bit manipulation functions
 *--
 */

#ifndef BIT_TWIDDLE_H_
#define BIT_TWIDDLE_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include <assert.h>
/*
 * Macros
 */
<<bit_twiddle.h: macros>>
/*
 * Data Type Declarations
 */
<<bit_twiddle.h: data type declarations>>
/*
 * Static Inline Functions
 */
<<bit_twiddle.h: inline functions>>

#endif /* BIT_TWIDDLE_H_ */
```

Chapter 14

System Register Manipulation

Stack Alignment

stwd_sp_align_8byte

Set the stack alignment indication to 8 byte alignment.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_sp_align_8byte(
    bool align)
{
    if (align) {
        BTWD_SET_REG_FIELD(&SCB->CCR, SCB_CCR_STKALIGN) ;
    } else {
        BTWD_CLEAR_REG_FIELD(&SCB->CCR, SCB_CCR_STKALIGN) ;
    }
}
```

Divide by Zero

stwd_enable_div_zero_trap

Enable or disable a trap on division by zero.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_enable_div_zero_trap(
    bool enable)
{
    if (enable) {
        BTWD_SET_REG_FIELD(&SCB->CCR, SCB_CCR_DIV_0_TRP) ;
    } else {
        BTWD_CLEAR_REG_FIELD(&SCB->CCR, SCB_CCR_DIV_0_TRP) ;
    }
}
```

Unaligned Memory Access

stwd_enable_unaligned_trap

Enable or disable a trap on unaligned memory access.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_enable_unaligned_trap(
    bool enable)
{
    if (enable) {
        BTWD_SET_REG_FIELD(&SCB->CCR, SCB_CCR_UNALIGN_TRP) ;
    } else {
        BTWD_CLEAR_REG_FIELD(&SCB->CCR, SCB_CCR_UNALIGN_TRP) ;
    }
}
```

Deep Sleep Mode

stwd_enable_deep_sleep

Enable or disable deep sleep mode in the processor.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_enable_deep_sleep(
    bool enable)
{
    if (enable) {
        BTWD_SET_REG_FIELD(&SCB->SCR, SCB_SCR_SLEEPDEEP) ;
    } else {
        BTWD_CLEAR_REG_FIELD(&SCB->SCR, SCB_SCR_SLEEPDEEP) ;
    }
}
```

Processor Execution Mode

stwd_is_thread_mode

Determine if the current execution mode is thread. Return `true` if executing in thread mode and `false` otherwise.

```
<<sys_twiddle.h: inline functions>>=
static inline bool
stwd_is_thread_mode(void)
{
    return __get_IPSR() == 0 ;
}
```

Running in Privileged Mode

stwd_is_priv_mode

Determine if the current execution is privileged. Return `true` if executing in privileged mode and `false` otherwise.

```
<<sys_twiddle.h: inline functions>>=
static inline bool
stwd_is_priv_mode(void)
{
    return (__get_IPSR() >= 0) || ((__get_CONTROL() & CONTROL_nPRIV_Msk) == 0) ;
}
```

Managing Exception Priorities

The Cortex-M4 architecture has flexible exception handling mechanisms. But there are a few fundamental operations which we need to control. An important one is to be able to execute code without taking an exception. This is necessary to control, for example, whenever an IRQ handler and background code share data.

The mechanism by which this is accomplished is related to the priority of the exception. Some exceptions have fixed priorities. These are exceptions which are fundamental to the processor execution:

- **Reset**
- **NMI** (Non-Maskable Interrupt)
- **HardFault**

Other exceptions, such as IRQ exceptions or system defined exceptions have a configurable priority. As a general rule, the code which has the highest priority is executed by the processor. If two exception have the same priority, it is first come - first served, *i.e.* an exception of the same priority as one which is executing will not preempt the running exception — to preempt the priority must be strictly higher. There are three special registers used to manage priorities at a course grain.

- **FAULTMASK** ⇒ prevents activation of all exceptions except **NMI**
- **PRIMASK** ⇒ prevents activation of all exceptions with configurable priority.
- **BASEPRI** ⇒ sets the minimum priority an exception must have to preempt execution.

Important

The numbers used to *encode* exception priority field values of the system registers are the opposite of the *order* of priority preemption. Also, the number of bits of priority encoding is variable. This can be a source of confusion. The highest priority is encoded as zero. The lowest priority is encoded as 255[a]. Just remember:

- high priority for preemption ⇒ smaller priority number
- low priority for preemption ⇒ larger priority number

[a] the actual number of priority steps available can vary depending upon the number of bits of priority a chip vendor chooses to implement

In most systems, we do *not* change the value of the **FAULTMASK** register. If we take a **HardFault** in the middle of something we usually want to know about it straight away. But there will be times when we want to prevent all IRQ's or we just want to insure some exceptions can be processed and others can't. Code that runs under conditions where preemption is prevented is usually referred to as a *critical section*. We wish to keep such code sections short, since the processor is prevented from responding to any interrupting condition and is committed to running the critical section to completion.

Preventing Preemption by Exceptions

stwd_begin_critical_section

Start a section of code where there will be no preemption by exceptions with configurable priorities, *e.g.* IRQ requests. Critical section manipulations are presumed to be paired, *i.e.* there is a matching *end* for each *begin*. To support such pairing and to allow the function to be invoked when a critical section is already in place, this function returns the current state of **PRIMASK**. It is **imperative** that the end of the critical section restore the returned **PRIMASK** value.

```
<<sys_twiddle.h: inline functions>>=
static inline uint32_t
stwd_begin_critical_section(void)
{
    uint32_t primask = __get_PRIMASK() ;
    __set_PRIMASK(UINT32_C(1)) ;
    return primask ;
}
```

stwd_end_critical_section

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_end_critical_section(
    uint32_t primask)
{
    __set_PRIMASK(primask) ;
}
```

Because it is important to pair beginning and ending a critical section, experienced programmers try to keep the begin / end operations lexically close to each other in the source file. This makes it more readily apparent that there is no path through the code which could fail to restore **PRIMASK** properly. Again, this is a case where we can use the pre-processor to do some typing for us.

EXEC_CRITICAL_SECTION

The `EXEC_CRITICAL_SECTION` macro expands to insure the code given by `section` executes within a critical section.

```
<<sys_twiddle.h: macros>>=
#define EXEC_CRITICAL_SECTION(section)                  \
{                                                       \
    uint32_t exc_mask = stwd_begin_critical_section() ; \
    {                                                   \
        section                                         \
    }                                                   \
    stwd_end_critical_section(exc_mask) ;               \
}
```

Partially Preventing Preemption

There are times when we wish to allow some exceptions of a higher priority to occur, but prevent lower priority ones from otherwise interfering with the execution. These cases can arise when using system exceptions together with interrupt exceptions.

Consider the following example. We can choose to use the **PendSV** exception as a means of triggering the synchronization between an IRQ handler and the background processing. In this usage, an IRQ handler sets the **PendSV** exception as pending when it wishes to signal to the background that the IRQ happened and there is additional work that needs to be done. The

background processing wishes to wait until there is a *safe* opportunity to handle the **PendSV** exception. For example, the **PendSV** handler and the background might share some data which could cause failures if the background were preempted arbitrarily. However when the background is running, other IRQ handlers need to be free to run and respond to their interrupts. This scheme for a simple scheduling mechanism. One way to accomplish this is:

- Set the **PendSV** exception priority to the lowest priority (highest priority number).
- Set the exception priorities for the IRQ's to a higher priority than **PendSV** (lower priority number).
- When the background execution cannot tolerate servicing **PendSV**, then the base priority of the background is set equal to that of the **PendSV** exception. This prevents the **PendSV** handler from running, but allows the other IRQ handlers to preempt background execution.

To handle this case, we construct functions to manipulate the **BASEPRI** register in a manner similar to the previous functions that worked on the **PRIMASK** register.

stwd_begin_priority_section

Start a section of code running at a base priority.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_begin_priority_section(
    uint32_t priority)
{
    uint32_t encoded_priority = NVIC_EncodePriority(PRIORITY_GROUP, priority, 0) ;  // ❶
    __set_BASEPRI(encoded_priority << (8U - __NVIC_PRIO_BITS)) ;
}
```

❶ This is tricky here. The encoded priority value is suitable to assign to interrupt priority registers in the NVIC using the CMSIS `NVIC_SetPriority` function. This function shifts the priority into the correct location within the interrupt priority register. Recall that the significant interrupt priority bits are the *high* bits. However, to set the `BASEPRI` register we have to perform that bit shift ourselves. So here we use `NVIC_EncodePriority` to encode the priority value correctly for the priority grouping we have set up, but have to shift it to the correct position before writing the `BASEPRI` register. It's subtle and an easy source of error.

stwd_end_priority_section

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_end_priority_section(void)
{
    __set_BASEPRI(0) ;
}
```

N.B. the `BASEPRI` register is always set to zero to end the priority section. The `BASEPRI` register serves effectively as an interrupt mask and a value of zero indicates that no masking is requested. The processor continues to run at the appropriate execution priority, *i.e.* an exception handler which has modified `BASEPRI` to exclude some exception continues to run at it configured priority after setting `BASEPRI` to zero.

Following the same reasoning as for critical sections, we give a macro to execute code in a priority section.

The `EXEC_PRIORITY_SECTION` macro expands to insure the code given by `section` executes within a priority section.

EXEC_PRIORITY_SECTION

```
<<sys_twiddle.h: macros>>=
#define EXEC_PRIORITY_SECTION(priority, section)                \
{                                                               \
    stwd_begin_priority_section(priority) ;                     \
    {                                                           \
        section                                                 \
    }                                                           \
    stwd_end_priority_section() ;                               \
}
```

Preventing Interrupt Preemption

Sometimes it is necessary to execute code where preemption by a particular interrupt is prevented.

stwd_begin_interrupt_section

Start a section of code inhibiting a particular interrupt. Returns `true` if the interrupt was enabled. At the end of the function, the interrupt is disabled at the NVIC.

```
<<sys_twiddle.h: inline functions>>=
static inline bool
stwd_begin_interrupt_section(
    IRQn_Type irq)
{
    uint32_t enable = NVIC_GetEnableIRQ(irq) ;
    NVIC_DisableIRQ(irq) ;
    return enable != 0 ;
}
```

stwd_end_interrupt_section

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_end_interrupt_section(
    IRQn_Type irq,
    bool enable)
{
    if (enable) {
        NVIC_EnableIRQ(irq) ;
    }
}
```

Following the same reasoning as for critical sections, we give a macro to execute code in a priority section.

The `EXEC_IRQ_SECTION` macro expands to insure the code given by `section` executes within a interrupt section.

EXEC_INTERRUPT_SECTION

```
<<sys_twiddle.h: macros>>=
#define EXEC_IRQ_SECTION(irq, section)                          \
{                                                               \
    bool enable = stwd_begin_interrupt_section(irq) ;           \
    {                                                           \
        section                                                 \
    }                                                           \
    stwd_end_interrupt_section(irq, enable) ;                   \
}
```

System Control of Floating Point

```
<<sys_twiddle.h: data types>>=
typedef enum {
    fpu_access_denied = 0,
    fpu_access_privileged = 1,
    fpu_access_full = 3
} STWD_fpu_access_mode ;            // ❶
```

❶ *N.B.* the enumerator values have been carefully chosen to match the requirements of the `CPACR` bit fields. A value of 2 is not an allowable configuration.

stwd_set_fpu_access

Enable access to the FPU.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_set_fpu_access(
    STWD_fpu_access_mode mode)
{
#   if defined(__FPU_USED) && (__FPU_USED == 1U)        // ❶

    /*
     * Oddly, the core CMSIS files have no definitions for these fields.
     */
    static unsigned const CP10_offset = 20 ;
    static unsigned const CP11_offset = 22 ;
    static unsigned const CPACR_field_width = 2 ;

    uint32_t mode_reg = SCB->CPACR ;

    mode_reg = btwd_field_insert(mode_reg, mode, btwd_mask(CPACR_field_width,  ↩
        CP10_offset),
            CP10_offset) ;
    mode_reg = btwd_field_insert(mode_reg, mode, btwd_mask(CPACR_field_width,  ↩
        CP11_offset),
            CP11_offset) ;

    SCB->CPACR = mode_reg ;
    __DSB() ;
    __ISB() ;

#   endif /* __FPU_USED */
}
```

❶ The `__FPU_USED` symbol is available in the CMSIS definitions and indicates there is an FPU and the compiler settings are such that floating point instructions are being generated, *cf.* `__FPU_PRESENT`.

stwd_is_exc_frame_basic

Check if an exception return value indicates that the exception frame is "basic" or contains FPU register values.

```
<<sys_twiddle.h: inline functions>>=
static inline bool
stwd_is_exc_frame_basic(
    uint32_t exc_return)
{
    return btwd_bit_test(exc_return, 4) ;       // ❶
}
```

❶ Bit 4 is 0 if the exception return is to unstack floating point registers.

Floating Point Stack Context

```
<<sys_twiddle.h: data types>>=
typedef enum {
    fpu_no_auto_stack = 0,
    fpu_auto_stack_always = 2,
    fpu_auto_stack_lazy = 3
} STWD_fpu_context_mode ;
```

stwd_set_fpu_stack_context

Controls the stacking of floating point registers during exception entry.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_set_fpu_stack_context(
    STWD_fpu_context_mode mode)
{
#   if defined(__FPU_USED) && (__FPU_USED == 1U)

    (void)btwd_write_reg_field(&FPU->FPCCR, mode,
            FPU_FPCCR_ASPEN_Msk | FPU_FPCCR_LSPEN_Msk,
            FPU_FPCCR_LSPEN_Pos) ;                          // ❶
    __DSB() ;
    __ISB() ;

#   endif /* __FPU_USED */
}
```

❶ This is a contrived scheme to treat the two contiguous control bits as as single field. The two bits in the `FPCCR` register need to be handled as a pair, but are defined in the CMSIS header file as individual control bits (which does match the ARM register specification). There are only three valid configuration values — `LSPEN = 1` and `ASPEN = 0` is an invalid configuration. Using the bitwise-or of the masks and the `LSPEN` position, which is smaller than the `ASPEN` position, we can fake a two bit register field.

stwd_is_fpu_context_active

Determines if there is an active floating point context by examining the CONTROL register.

```
<<sys_twiddle.h: inline functions>>=
static inline bool
stwd_is_fpu_context_active(void)
{
    uint32_t control = __get_CONTROL() ;
    return BTWD_FIELD_TEST(control, CONTROL_FPCA) ;
}
```

System Exception Handler Functions

stwd_enable_bus_fault

Enable or disable the system bus fault exception.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_enable_bus_fault(
    bool enable)
{
    if (enable) {
        BTWD_SET_REG_FIELD(&SCB->SHCSR, SCB_SHCSR_BUSFAULTENA) ;
    } else {
        BTWD_CLEAR_REG_FIELD(&SCB->SHCSR, SCB_SHCSR_BUSFAULTENA) ;
    }
}
```

stwd_enable_usage_fault

Enable or disable the system usage fault exception.

```
<<sys_twiddle.h: inline functions>>=
static inline void
stwd_enable_usage_fault(
    bool enable)
{
    if (enable) {
        BTWD_SET_REG_FIELD(&SCB->SHCSR, SCB_SHCSR_USGFAULTENA) ;
    } else {
        BTWD_CLEAR_REG_FIELD(&SCB->SHCSR, SCB_SHCSR_USGFAULTENA) ;
    }
}
```

Debugger Functions

stwd_debug_enabled

Check if debugging is enable.

```
<<sys_twiddle.h: inline functions>>=
static inline bool
stwd_debug_enabled(void)
{
    return btwd_test_reg_field(&CoreDebug->DHCSR, CoreDebug_DHCSR_C_DEBUGEN_Msk) ;
}
```

stwd_debugmon_enabled

Check if the debug monitor exception is enabled.

```
<<sys_twiddle.h: inline functions>>=
static inline bool
stwd_debugmon_enabled(void)
{
    return btwd_test_reg_field(&CoreDebug->DEMCR, CoreDebug_DEMCR_MON_EN_Msk) ;
}
```

Code Layout

Sys Twiddle Include File Layout

Sys twiddle include file

```
<<sys_twiddle.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   sys_twiddle.h -- bit manipulation functions for system registers
 *--
 */

#ifndef SYS_TWIDDLE_H_
#define SYS_TWIDDLE_H_

/*
 * Include files
 */
#include <stdbool.h>
#include <stdint.h>
#include "apollo3.h"
#include "bit_twiddle.h"
#include "exc_priority.h"
/*
 * Macros
 */
<<sys_twiddle.h: macros>>
/*
 * Data types
 */
<<sys_twiddle.h: data types>>
/*
 * Static inline functions
 */
<<sys_twiddle.h: static inline functions>>
/*
 * Inline functions
 */
<<sys_twiddle.h: inline functions>>

#endif /* SYS_TWIDDLE_H_ */
```

Chapter 15

Device Register Manipulation

Fault Address Registers

dtwd_enable_fault_capture

Enable additional fault addresses registers provided by the Apollo 3.

```
<<dev_twiddle: static inline functions>>=
static inline bool
dtwd_enable_fault_capture(
    bool enable)
{
    uint32_t capture_enable = MCUCTRL->FAULTCAPTUREEN ;
    MCUCTRL->FAULTCAPTUREEN = enable ? 1 : 0 ;
    return capture_enable != 0 ;
}
```

enable	If `true` then additional fault address capture is enabled. If `false` then additional fault address capture is disabled.

Timestamps

dtwd_get_timestamp

Get the value of the system counter to us as a timestamp. The returned number is a Q49.15 unsigned value in units of seconds.

```
<<dev_twiddle: external function declarations>>=
extern uint64_t
dtwd_get_timestamp(void) ;
```

Implementation

```
<<dev_twiddle: external function definitions>>=
uint64_t
dtwd_get_timestamp(void)
{
    uint32_t cntr1 ;
```

```
    uint32_t cntr2 ;
    uint32_t snvr0 ;

    uint32_t loop = 0 ;
    do {
        cntr1 = CTIMER->STTMR ;
        snvr0 = CTIMER->SNVR[0] ;
        cntr2 = CTIMER->STTMR ;
        if (cntr1 == cntr2) {
            break ;
        }
    } while (++loop < 3) ;                                  // ❶

    return ((uint64_t)snvr0 << 32) | (uint64_t)cntr1 ;
}
```

❶ Try to account for a roll over just as we are reading the timer register. We suspect, but it not clear from the data sheet, that the SNVR[0-3] registers roll over with the timer. So we read the timer twice to insure that there was no roll over before we read the SNVR0 register. We're will to try this for 3 times (an arbitrary number but greater than 2) before giving up and taking whatever we found.

RTC Counters

dtwd_get_rtc_counters

Get the value of the RTC counters. The values are returned by reference.

```
<<dev_twiddle: external function declarations>>=
extern bool
dtwd_get_rtc_counters(
    uint32_t *const lower_ref,
    uint32_t *const upper_ref) ;
```

Places the values of upper and low RTC counter registers into the memory objects pointed to by `upper_ref` and `lower_ref`, respectively. Returns `true` if the counters were read successfully and `false` otherwise.

Implementation

```
<<dev_twiddle: external function definitions>>=
bool
dtwd_get_rtc_counters(
    uint32_t *const lower_ref,
    uint32_t *const upper_ref)
{
    unsigned tries = 0 ;
    uint32_t lower = RTC->CTRLOW ;
    uint32_t upper = RTC->CTRUP ;

    while (BTWD_FIELD_TEST(upper, RTC_CTRUP_CTERR) && ++tries < 3) {     // ❶
        lower = RTC->CTRLOW ;
        upper = RTC->CTRUP ;
    }

    if (lower_ref != NULL) {
        *lower_ref = lower ;
    }
    if (upper_ref != NULL) {
        *upper_ref = upper ;
```

```
    }

    return !BTWD_FIELD_TEST(upper, RTC_CTRUP_CTERR) ;
}
```

❶ Reading the time from the RTC requires two register reads. The peripheral “freezes” the upper register value when the lower one is read. On the off chance that an interrupt occurs after the lower register read but before the upper register read, the data between the two reads could be inconsistent. The peripheral diagnoses this situation using an error bit in the upper register to indicate that the read attempt was not successful. The retry count guards against the remote possibility of an infinite loop.

dtwd_in_process_stack

```
<<dev_twiddle: static inline functions>>=
static inline bool
dtwd_in_process_stack(
    uint8_t *const ptr)
{
    uint64_t *ptr_64 = (uint64_t *)ptr ;
    return ptr_64 >= &__process_stack_limit__ && ptr_64 < &__process_stack_top__ ;
}
```

Returns `true` if `ptr` is within the memory region allocated to the process stack and `false` otherwise.

Pointers into the process stack

These two functions determine whether a pointer value points into a stack area. In some cases, foreground device control code needs to disallow pointer values which are located in the process stack area. This is the case when buffer pointers are passed in background requests which are fulfilled asynchronously. For this case, a stack address is most likely invalid before the operation can complete.

Code Layout

Dev Twiddle Include File Layout

Dev twiddle include file

```
<<dev_twiddle.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   dev_twiddle.h -- bit manipulation functions for device peripheral registers
 *--
 */

#ifndef DEV_TWIDDLE_H_
#define DEV_TWIDDLE_H_

/*
```

```
 * Include files
 */
#include <stdint.h>
#include "apollo3.h"
#include "bit_twiddle.h"
#include "linker_symbols.h"
/*
 * Macros
 */
<<dev_twiddle.h: macros>>
/*
 * Data types
 */
<<dev_twiddle: data types>>
/*
 * Static inline functions
 */
<<dev_twiddle: static inline functions>>
/*
 * External Function Declarations
 */
<<dev_twiddle: external function declarations>>

#endif /* DEV_TWIDDLE_H_ */
```

Dev Twiddle Code File Layout

Dev twiddle code file

```
<<dev_twiddle.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   dev_twiddle.c -- bit manipulation functions for device peripheral registers
 *--
 */

/*
 * Include files
 */
#include "dev_twiddle.h"
/*
 * External Functions
 */
<<dev_twiddle: external function definitions>>
```

Chapter 16

System Information

In this section, a set of functions is developed to format information which is available from the system. This is essentially meta-information about the system which can be used to support configuration management. The design goal for these functions is to build a set of specializations to format the system information which is build upon `snprintf()`. Rather than print directly to an I/O device, human readable strings are created in a regular format. This approach makes using this information for devices other than a console terminal easier. If console output is needed, as was shown in Part II, then `printf()` or `fputs()` is available.

Prototypes for the standard I/O functions and some string functions are needed.

```
<<sysinfo: include files>>=
#include <stdio.h>
#include <string.h>
```

Build ID

A build ID was placed into the executable in Part I.

Being able to precisely determine what code you have in flash memory is important, perhaps not initially, but later on it will save you time. The idea and basic design for this capability was cribbed directly from an interrupt.memfault.com article.

The underlying idea uses the ability for the GCC linker to compute (or simply accept) a unique value for an executable. It calls such an identifier a, build ID. In this section a function to retrieve the build ID is provided. There are several different build ID types available. The default one is a SHA1 hash. However, the build ID option will also take a specified hexadecimal number. This opens the possibility of using a hash value from a source control system to mark an executable as being derived from a particular commit of the source. This assumes that the build process is repeatable, something that must be well tested to insure.

```
<<sysinfo: include files>>=
#include "linker_symbols.h"
```

sysi_build_id

The `sysi_build_id` function returns a reference to and the size of a memory object which holds a unique identifier for the program executable currently in flash.

```
<<sysinfo: external prototypes>>=
extern void sysi_build_id(uint8_t const **const id_ref, uint32_t *const size_ref) ;
```

id_ref	A pointer to a pointer used to return the address of the array which holds the system build ID. If `id_ref` is `NULL`, then no system ID array pointer is returned.
size_ref	A pointer to a variable where the number of bytes of data contained in the system build ID is placed. If `size_ref` is `NULL` then no size information is returned.

sysi_build_id Implementation

```
<<sysinfo: external functions>>=
void
sysi_build_id(
    uint8_t const **const id_ref,
    uint32_t *const size_ref)
{
    if (id_ref != NULL) {
        size_t const desc_offset =
                btwd_align(SystemBuildIDNote.namesz, sizeof(uint32_t)) ;     // ❶
        *id_ref = &SystemBuildIDNote.data[desc_offset] ;
    }
    if (size_ref != NULL) {
        *size_ref = SystemBuildIDNote.descsz ;
    }
}
```

❶ This bit of alignment twiddling accounts for padding that might be included in the `name` field but which is *not* included in the `namesz` field value. In this case, the `name` field is padded to 32 bits (so that the description field is aligned to 32 bits), but the `namesz` field does _not_account for any padding. What can you say? This type of stuff happens.

sysi_format_build_id

Returns the number of characters written to `bid_buf`, excluding the `NUL` terminator. Negative return values indicate an error. Return values greater than or equal to `capacity` indicate that the output was truncated due to lack of space.

```
<<sysinfo: external prototypes>>=
extern int
sysi_format_build_id(
    size_t capacity,
    char bid_buf[capacity]) ;
```

capacity	The number of bytes pointed to by `bid_buf`.
bid_buf	A pointer to a memory object of `capacity` number of bytes where the formatted build ID string is written. The returned string is `NUL` terminated.

To help estimate the amount of memory the formatted output will take, a default length is defined. This default length is accounts for the default build ID length the linker uses, the set of interspersed underscore ("_") characters used to increase readability, and the terminating `NUL` character.

```
<<sysinfo: constants>>=
#define SYSINFO_BUILD_ID_DEFAULT_LEN    (20 * 2 + 10 + 1)
```

A function to locate the build ID data has already been written. Recall that this information is placed in the executable by the linker at our direction.

```
<<sysinfo: include files>>=
#include "system_apollo3.h"
```

The implementation of `sysi_format_build_id()` factors the computation into two parts, obtaining the build ID information and formatting it as a series of hexadecimal numbers with an underscore placed every four characters.

Implementation

```
<<sysinfo: external functions>>=
int
sysi_format_build_id(
    size_t capacity,
    char bid_buf[capacity])
{
    uint8_t const *build_id_bytes ;
    uint32_t build_id_len ;
    sysi_build_id(&build_id_bytes, &build_id_len) ;

    assert(build_id_bytes != NULL) ;
    assert(build_id_len != 0) ;

    return sysi_format_bytes_by_twos(capacity, bid_buf, build_id_len, build_id_bytes) ;
}
```

Common Formatting

To aid in reading long sequences of hexadecimal numbers, an array of byte data as 16-bit quantities is formatted and an underscore character is placed between each grouping of two formatted data bytes. For example, instead of:

```
0123456789abcdef
```

underscores are preferred for readabilitiy.

```
0123_4567_89ab_cdef
```

```
<<sysinfo: static functions>>=
static int
sysi_format_bytes_by_twos(
    size_t output_len,
    char output[output_len],
    size_t src_len,
    uint8_t const src[src_len])
{
    if (output_len == 0 || output == NULL) {                    // ❶
        return 0 ;
    }

    int formatted_len = 0 ;

    while (src_len != 0) {
        int nbytes ;

        if (src_len > 1) {
```

```
            nbytes = snprintf(output, output_len, "%02x%02x_", src[0], src[1]) ;
            src_len -= 2 ;
            src += 2 ;
        } else {
            nbytes = snprintf(output, output_len, "%02x", src[0]) ;
            src_len -= 1 ;
            src += 1 ;
        }

        if (nbytes < 0) {
            return nbytes ;
        }
        formatted_len += nbytes ;
        if (nbytes >= output_len) {
            break ;
        }
        output += nbytes ;
        output_len -= nbytes ;
    }

    if (formatted_len > 0 && *(output - 1) == '_') {                // ❷
        *(output - 1) = '\0' ;
        formatted_len -= 1 ;
    }

    return formatted_len ;
}
```

❶ Dispense with the boundary case immediately.

❷ Remove any trailing underscore. `output` points to the `NUL` terminating character.

Chip ID

Two hardware registers hold a 64-bit number which is unique to each Apollo 3 chip.

sysi_format_chip_id

Returns the number of characters written to `cid_buf`, excluding the `NUL` terminator. Negative return values indicate an error. Return values greater than or equal to `capacity` indicate that the output was truncated due to lack of space.

```
<<sysinfo: external prototypes>>=
extern int
sysi_format_chip_id(
    size_t capacity,
    char cid_buf[capacity]) ;
```

capacity	The number of bytes pointed to by `cid_buf`.
cid_buf	A pointer to a memory object of `capacity` number of bytes where the formatted chip ID string is written. The returned string is `NUL` terminated.

The length of the formatted output accounts for writing two 32-bit values with an underscore to separate them and the `NUL` terminator.

```
<<sysinfo: constants>>=
#define SYSINFO_CHIP_ID_DEFAULT_LEN     (8 * 2 + 1 + 1)
```

The chip ID values are fetched directly from their hardware registers. The CMSIS header file locates the registers.

```
<<sysinfo: include files>>=
#include "apollo3.h"
```

Implementation

```
<<sysinfo: external functions>>=
int
sysi_format_chip_id(
    size_t capacity,
    char cid_buf[capacity])
{
    return snprintf(cid_buf, capacity, "%08" PRIx32 "_%08" PRIx32,
            MCUCTRL->CHIPID1, MCUCTRL->CHIPID0) ;                    // ❶
}
```

❶ The chip ID is treated as a little endian value.

The definitions of the format strings for sized integers are contained in a standard header file.

```
<<sysinfo: include files>>=
#include <inttypes.h>
```

Chip Info

The `CHIPPN` register provides several pieces of information about the chip such as it part number, memory sizes, etc.

sysi_format_chip_info

Returns the number of characters written to `info_buf`, excluding the `NUL` terminator. Negative return values indicate an error. Return values greater than or equal to `capacity` indicate that the output was truncated due to lack of space.

```
<<sysinfo: external prototypes>>=
extern int
sysi_format_chip_info(
    size_t capacity,
    char info_buf[capacity]) ;
```

capacity	The number of bytes pointed to by `info_buf`.
info_buf	A pointer to a memory object of `capacity` number of bytes where the formatted chip ID string is written. The returned string is `NUL` terminated.

```
<<sysinfo: constants>>=
#define SYSINFO_CHIP_INFO_DEFAULT_LEN    100        /* ❶ */
```

❶ Empirically determined value.

Data Driven Register Formatting

It turns out that picking apart the fields of a hardware register and formatting the values of the fields is a common calculation to make. Hardware designers encode registers as bit fields and people need a readable interpretation of the contents. Some infrastructure in the form of descriptive data and a function that uses the data to control the outcome of the formatting is useful.

The fields are described within a hardware register by giving each field a name and the definition of the field. The field is defined using CMSIS conventions of specifying a maks for the field and the offset of the start of a field. These values are directly available from the CMSIS header file, `apollo3.h` in our case.

Field values are interpreted as either:

- A simple integer value.
- An enumeration value which is formatted into a string.

Many enumeration values for hardware registers are encoded as a set of sequential integers. These are handled by using the field value as an index into an array of character strings. For those enumeration values which are *not* encoded conveniently as array indices, a function is invoked to handle the case. Our treatment of field values must be able to distinguish between three cases:

1. The field value can be used as an array index.
2. The field value encodes an enumerator but in a form *not* suitable as an array index.
3. The field value is treated numerically.

These cases are encoded as follows:

- A "C" `enum` is used to distinguish between field which can be used as an index and those where a function must be supplied to custom map the field value to a string.
- For those field values using a function, a `NULL` function pointer is interpreted as a request to format the field value as an ordinary number.

The following data structure provides the mechanism to implement the design.

```
<<sysinfo: data type declarations>>=
typedef struct {
    char const *field_name ;
    unsigned field_mask ;
    unsigned field_pos ;
    enum {
        sfd_Array,
        sfd_Func,
    } value_desc ;                                  // ❶
    union {
        struct {
            size_t name_count ;
            char const *const *name_array ;
        } ;
        char const *(*name_func)(uint32_t index) ;
    } ;                                             // ❷
} SYSI_FieldDesc ;
```

❶ This is what a *discriminated union* looks like in "C". Tagging must be provided explicitly.

❷ Here an anonymous union and an anonymous structure are used to remove extraneous member naming for the field description structure. This is a standard `C11` language feature.

```
<<sysinfo: data type declarations>>=
typedef struct {
    char const *title ;
    size_t field_count ;
    SYSI_FieldDesc const *fields ;
} SYSI_Reg_Desc ;
```

The formatting for hardware register is then determined by an array of `SYSI_FieldDesc` elements, one for each field in the register. The following function performs the formatting according to the descriptor information given as parameters.

Some bit twiddling is required to help extract the fields from a register value.

```
<<sysinfo: include files>>=
#include "bit_twiddle.h"
```

```
<<sysinfo: static functions>>=
static int
sysi_format_register_fields(
    uint32_t hdwr_register,
    SYSI_Reg_Desc const *const reg_desc,
    size_t capacity,
    char info_buf[capacity])
{
    static char const leader[] = " " ;
    static char const trailer[] = "," ;

    int formatted_len = 0 ;

    int nbytes = snprintf(info_buf, capacity, "%s: 0x%08" PRIx32 "\n",
            reg_desc->title, hdwr_register) ;
    if (nbytes < 0) {
        return nbytes ;
    }
    formatted_len += nbytes ;
    if (nbytes >= capacity) {
        return formatted_len ;
    }
    info_buf += nbytes ;
    capacity -= nbytes ;

    static char const indent[] = "    " ;
    if (reg_desc->field_count != 0 && capacity > sizeof(indent)) {
        strcpy(info_buf, indent) ;
        info_buf += strlen(indent) ;
        capacity -= strlen(indent) ;
    }

    SYSI_FieldDesc const *const end_desc = reg_desc->fields + reg_desc->field_count ;
    for (SYSI_FieldDesc const *desc_iter = reg_desc->fields ;
            desc_iter < end_desc ; desc_iter++) {
        uint32_t field_value = btwd_field_extract(hdwr_register, desc_iter->field_mask,
                desc_iter->field_pos) ;

        switch(desc_iter->value_desc) {
        case sfd_Array:
            assert(field_value < desc_iter->name_count) ;

            if (field_value < desc_iter->name_count) {
                nbytes = snprintf(info_buf, capacity, "%s%s: %s%s",
                        leader, desc_iter->field_name,
                        desc_iter->name_array[field_value], trailer) ;
            } else {
                nbytes = snprintf(info_buf, capacity,
                        "%s%s: bad_value(%" PRIu32 ")%s",
                        leader, desc_iter->field_name, field_value, trailer) ;
            }
            break ;

        case sfd_Func:
```

```
            if (desc_iter->name_func == NULL) {
                nbytes = snprintf(info_buf, capacity, "%s%s: %" PRIu32 "%s",
                        leader, desc_iter->field_name, field_value, trailer) ;
            } else {
                nbytes = snprintf(info_buf, capacity, "%s%s: %s%s",
                        leader, desc_iter->field_name,
                        desc_iter->name_func(field_value), trailer) ;
            }
            break ;
            /*
             * N.B. no default case. If the switch falls through, the
             * field is simply skipped.
             */
        }

        if (nbytes < 0) {
            return nbytes ;
        }
        formatted_len += nbytes ;
        if (nbytes >= capacity) {
            break ;
        }
        info_buf += nbytes ;
        capacity -= nbytes ;
    }

    if (formatted_len > 0 && *(info_buf - 1) == *trailer) {
        *(info_buf - 1) = '\0' ;
        formatted_len -= 1 ;
    }

    return formatted_len ;
}
```

Armed with a data driven formatting function, to implement `sysi_format_chip_info()`, the descriptive information is supplied for the `CHIPPN` register.

Implementation

```
<<sysinfo: external functions>>=
int
sysi_format_chip_info(
    size_t capacity,
    char info_buf[capacity])
{
    <<chip info: chippn descriptors>>

    static SYSI_Reg_Desc const chippn_reg_desc = {
        .title = "CHIPPN",
        .field_count = COUNTOF(chippn_descriptors),
        .fields = chippn_descriptors,
    } ;

    return sysi_format_register_fields(MCUCTRL->CHIPPN, &chippn_reg_desc,
            capacity, info_buf) ;
}
```

Supplying the required data is a tedious operation of laying out a set of initialized variables containing the required descriptor values. Fortunately, many of the values are available from the CMSIS header file.

Starting with those fields whose values are suitable for use as an array index, the mapping of field value to enumerator string is defined as an array of pointers to character strings.

There are six fields in the `CHIPPN` register which have values that are enumerated and encoded suitably for array indices. The mapping arrays are listed below.

```
<<chip info: chippn descriptors>>=
static char const *const flash_size_names[] = {
    [MCUCTRL_CHIPPN_FLASHSIZE_16KB] = "16KB",           // ❶
    [MCUCTRL_CHIPPN_FLASHSIZE_32KB] = "32KB",
    [MCUCTRL_CHIPPN_FLASHSIZE_64KB] = "64KB",
    [MCUCTRL_CHIPPN_FLASHSIZE_128KB] = "128KB",
    [MCUCTRL_CHIPPN_FLASHSIZE_256KB] = "256KB",
    [MCUCTRL_CHIPPN_FLASHSIZE_512KB] = "512KB",
    [MCUCTRL_CHIPPN_FLASHSIZE_1M] = "1M",
    [MCUCTRL_CHIPPN_FLASHSIZE_2M] = "2M",
} ;
```

❶ Note the use of *designated initializers*. This insures that the array elements are in the correct order and since the order is specified by a pre-processor symbol obtained from the CMSIS headers, any changes are relatively immune to the enumerator encodings.

```
<<chip info: chippn descriptors>>=
static char const *const sram_size_names[] = {
    [MCUCTRL_CHIPPN_SRAMSIZE_16KB] = "16KB",
    [MCUCTRL_CHIPPN_SRAMSIZE_32KB] = "32KB",
    [MCUCTRL_CHIPPN_SRAMSIZE_64KB] = "64KB",
    [MCUCTRL_CHIPPN_SRAMSIZE_128KB] = "128KB",
    [MCUCTRL_CHIPPN_SRAMSIZE_256KB] = "256KB",
    [MCUCTRL_CHIPPN_SRAMSIZE_512KB] = "512KB",
    [MCUCTRL_CHIPPN_SRAMSIZE_1M] = "1M",
    [MCUCTRL_CHIPPN_SRAMSIZE_384KB] = "384KB",
    [MCUCTRL_CHIPPN_SRAMSIZE_768KB] = "768KB",
} ;
```

```
<<chip info: chippn descriptors>>=
static char const *const major_rev_names[] = {
    [0] = "unknown",
    [MCUCTRL_CHIPPN_MAJOR_REV_REV_A] = "A",
    [MCUCTRL_CHIPPN_MAJOR_REV_REV_B] = "B",
} ;
```

```
<<chip info: chippn descriptors>>=
static char const *const minor_rev_names[] = {
    [0] = "unknown",                                // ❶
    [MCUCTRL_CHIPPN_MINOR_REV_REV_0] = "0",
} ;
```

❶ The counting is a little strange here. Yes, 1 really does mean "0".

```
<<chip info: chippn descriptors>>=
static char const *const pkg_names[] = {
    [MCUCTRL_CHIPPN_PKG_SIP] = "SIP",
    [MCUCTRL_CHIPPN_PKG_QFN] = "QFN",
    [MCUCTRL_CHIPPN_PKG_BGA] = "BGA",
    [MCUCTRL_CHIPPN_PKG_CSP] = "CSP",
} ;
```

```
<<chip info: chippn descriptors>>=
static char const *const pin_names[] = {
    [MCUCTRL_CHIPPN_PINS_25_PINS] = "25",
    [MCUCTRL_CHIPPN_PINS_49_PINS] = "49",
    [MCUCTRL_CHIPPN_PINS_64_PINS] = "64",
    [MCUCTRL_CHIPPN_PINS_81_PINS] = "81",
    [MCUCTRL_CHIPPN_PINS_104_PINS] = "104",
} ;
```

One of the fields in `CHIPPN` is encoded such that it is simplest to construct a function to map the value to a string.

```
<<sysinfo: static functions>>=
static char const *
sysi_format_part_num(
    uint32_t part_num)
{
    char const *num_string ;

    switch (part_num) {
    case MCUCTRL_CHIPPN_PARTNUM_APOLLO3:
        num_string = "Apollo3" ;
        break ;

    case MCUCTRL_CHIPPN_PARTNUM_APOLLO2:
        num_string = "Apollo2" ;
        break ;

    case MCUCTRL_CHIPPN_PARTNUM_APOLLO:
        num_string = "Apollo" ;
        break ;

    default:
        num_string = "unknown" ;
        break ;
    }

    return num_string ;
}
```

Finally, the array/function mappings which describe each field in the `CHIPPN` register can be defined.

```
<<chip info: chippn descriptors>>=
static SYSI_FieldDesc const chippn_descriptors[] = {
    {
        .field_name = "PN",
        .field_mask = MCUCTRL_CHIPPN_PARTNUM_Msk,
        .field_pos = MCUCTRL_CHIPPN_PARTNUM_Pos,
        .value_desc = sfd_Func,
        .name_func = sysi_format_part_num,
    },
    {
        .field_name = "FLASHSIZE",
        .field_mask = MCUCTRL_CHIPPN_FLASHSIZE_Msk,
        .field_pos = MCUCTRL_CHIPPN_FLASHSIZE_Pos,
        .value_desc = sfd_Array,
        .name_count = COUNTOF(flash_size_names),
        .name_array = flash_size_names,
    },
    {
        .field_name = "SRAMSIZE",
        .field_mask = MCUCTRL_CHIPPN_SRAMSIZE_Msk,
```

```
        .field_pos = MCUCTRL_CHIPPN_SRAMSIZE_Pos,
        .value_desc = sfd_Array,
        .name_count = COUNTOF(sram_size_names),
        .name_array = sram_size_names,
    },
    {
        .field_name = "MAJOR_REV",
        .field_mask = MCUCTRL_CHIPPN_MAJOR_REV_Msk,
        .field_pos = MCUCTRL_CHIPPN_MAJOR_REV_Pos,
        .value_desc = sfd_Array,
        .name_count = COUNTOF(major_rev_names),
        .name_array = major_rev_names,
    },
    {
        .field_name = "MINOR_REV",
        .field_mask = MCUCTRL_CHIPPN_MINOR_REV_Msk,
        .field_pos = MCUCTRL_CHIPPN_MINOR_REV_Pos,
        .value_desc = sfd_Array,
        .name_count = COUNTOF(minor_rev_names),
        .name_array = minor_rev_names,
    },
    {
        .field_name = "PKG",
        .field_mask = MCUCTRL_CHIPPN_PKG_Msk,
        .field_pos = MCUCTRL_CHIPPN_PKG_Pos,
        .value_desc = sfd_Array,
        .name_count = COUNTOF(pkg_names),
        .name_array = pkg_names,
    },
    {
        .field_name = "PINS",
        .field_mask = MCUCTRL_CHIPPN_PINS_Msk,
        .field_pos = MCUCTRL_CHIPPN_PINS_Pos,
        .value_desc = sfd_Array,
        .name_count = COUNTOF(pin_names),
        .name_array = pin_names,
    },
    {
        .field_name = "TEMP",
        .field_mask = MCUCTRL_CHIPPN_TEMP_Msk,
        .field_pos = MCUCTRL_CHIPPN_TEMP_Pos,
        .value_desc = sfd_Func,
        .name_func = NULL,
    },
    {
        .field_name = "QUAL",
        .field_mask = MCUCTRL_CHIPPN_QUAL_Msk,
        .field_pos = MCUCTRL_CHIPPN_QUAL_Pos,
        .value_desc = sfd_Func,
        .name_func = NULL,
    },
} ;
```

System Information Code Layout

Sysinfo include file

```
<<sysinfo.h>>=
<<edit warning>>
```

```
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Ambiq Apollo 3 System Information Functions
 *--
 */
#ifndef SYSINFO_H_
#define SYSINFO_H_

/*
 * Include files
 */
#include <stddef.h>
/*
 * Constants
 */
<<sysinfo: constants>>
/*
 * External Functions
 */
<<sysinfo: external prototypes>>

#endif /* SYSINFO_H_ */
```

System information code file

```
<<sysinfo.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Code to Models
 *
 * Module:
 *   Ambiq Apollo 3 System Information Functions
 *--
 */

/*
 * Include files
 */
#include "useful.h"
<<sysinfo: include files>>
#include "sysinfo.h"
/*
 * Data Type Declarations
 */
<<sysinfo: data type declarations>>
/*
 * Static Data
 */
<<sysinfo: static data>>
/*
 * Static Functions
 */
<<sysinfo: static functions>>
/*
```

```
 * External Functions
 */
<<sysinfo: external functions>>
```

Chapter 17

Synchronous State Machines

The implementation shown here is in the C11 dialect of the "C" programming language.

The following sections step through each component of the state machine mechanism to describe its implementation.

Defining States

We define a small integer to hold the value of a state. These state machines are not intended to be large. We intend to supply values for the state through an enumeration.

```
<<sync sm: interface data type declarations>>=
typedef int8_t SSM_State ;
```

To support the notion of an ignored event or a unexpected event, we define negative valued constants. These constants are non-transitioning state indicator.

```
<<sync sm: interface constants>>=
#define SSM_State_IG        (-1)
#define SSM_State_CH        (-2)
```

A transition may be *ignored* (`SSM_State_IG`) or be deemed a logical impossibility (*i.e.* can't happen, `SSM_State_CH`).

Each state has an associated activity function.

```
<<sync sm: interface data type declarations>>=
typedef void (*SSM_Activity)(struct ssm_machine *const machine) ;
```

We need a forward reference to the `ssm_machine` structure since we have not yet defined what a state machine is.

```
<<sync sm: interface forward references>>=
struct ssm_machine ;
```

Note the argument to the function is a pointer to the state machine. State activities are expected to recover the type of any larger containing structure as part of the processing. To help recovering the type of a containing structure, we provide and implementation of the venerable `CONTAINER_OF` macro.

```
<<sync sm: interface macros>>=
#ifndef CONTAINER_OF

#   define CONTAINER_OF(ptr, type, member)\
        (type *)((uintptr_t)ptr - offsetof(type, member))

#endif /* CONTAINER_OF */
```

Defining Events

Events are encoded as small unsigned numbers. Like states, we intend to supply the values using an enumeration.

```
<<sync sm: interface data type declarations>>=
typedef uint8_t SSM_Event ;
```

Queuing Events

Each state machine, *i.e.* each instance using a given state model, has its own event queue. Once the state machine dispatch has begun, it continues until the event queue is empty. Consequently, there is no need for large event queues. Since there are no event parameters, a simple array of `SSM_Event` values is sufficient storage for the event queue.

```
<<sync sm: interface data type declarations>>=
typedef SSM_Event SSM_EventPool[SSM_DEFAULT_EVENT_POOL_SIZE] ;
```

We choose a default value for the event pool size which allows for a signaled event, a self-directed event, an empty slot to detect overflow and one slot for margin. Turns out four is also a convenient number for memory usage and alignment.

```
<<sync sm: interface macros>>=
#ifndef SSM_DEFAULT_EVENT_POOL_SIZE
#   define SSM_DEFAULT_EVENT_POOL_SIZE     4
#endif /* SSM_DEFAULT_EVENT_POOL_SIZE */
```

The event queue consists of the storage pool for the events and sufficient bookkeeping to track a circular queue.

```
<<sync sm: interface data type declarations>>=
typedef struct {
    SSM_EventPool pool ;
    SSM_Event *head ;
    SSM_Event *tail ;
    SSM_Event *const end ;
} SSM_EventQueue ;
```

There are four supported operations on the event queue:

1. Insert at the tail of the queue. This is the normal event signaling operation.
2. Insert at the head of the queue. This operation is used to implement the self-directed events necessary for transitory states.
3. Remove from the head of the queue. This operation is used to obtain an event for dispatch.
4. Initialize the queue to its empty state.

The event queue operations implement a conventional circular queue. Overflow is detected and is an error. Underflow is not an error and the condition can be used for flow control in the event dispatch operations.

SSM_insertAtTail

Insert `event` at the *tail* of the `queue`. Returns `true` if the insertion took place and `false` otherwise.

```
<<sync sm: static functions>>=
static bool
SSM_insertAtTail(
    SSM_EventQueue *queue,
    SSM_Event event)
{
    SSM_Event *next_tail = queue->tail + 1 ;
    if (next_tail >= queue->end) {
        next_tail = queue->pool ;
    }
    assert(next_tail != queue->head) ;
    if (next_tail == queue->head) {             // ❶
        return false ;
    }

    *queue->tail = event ;
    queue->tail = next_tail ;
    return true ;
}
```

❶ Overflow is detected when a potential insertion would cause the head and tail pointers to be equal. Equal head and tail pointers define the queue empty condition. Overflow is detected as the situation where the queue would appear empty if an insertion took place.

queue	a pointer to the queue into which the event is inserted.
event	the event to insert.

The implementation of inserting at the head of the queue is similar to that of inserting at the tail. Inserting at the head involves moving *backwards* in the queue.

SSM_insertAtHead

Insert `event` at the *head* of the `queue`. Returns `true` if the insertion took place and `false` otherwise.

```
<<sync sm: static functions>>=
static bool
SSM_insertAtHead(
    SSM_EventQueue *queue,
    SSM_Event event)
{
    SSM_Event *new_head = queue->head - 1 ;
    if (new_head < queue->pool) {
        new_head = queue->end - 1 ;
    }
    assert(new_head != queue->tail) ;
    if (new_head == queue->tail) {
        return false ;
    }

    queue->head = new_head ;
    *queue->head = event ;
    return true ;
}
```

queue	a pointer to the queue into which the event is inserted.
event	the event to insert.

An event queue is only accessed from its *head*

SSM_removeFromHead

```
<<sync sm: static functions>>=
static int
SSM_removeFromHead(
    SSM_EventQueue *queue)
{
    int event = -1 ;

    if (queue->head != queue->tail) {
        event = *queue->head ;

        queue->head += 1 ;
        if (queue->head >= queue->end) {
            queue->head = queue->pool ;
        }
    }

    return event ;
}
```

queue	a pointer to the queue from which the event is removed.

SSM_initEventQueue

```
<<sync sm: static inline functions>>=
static inline void
SSM_initEventQueue(
    SSM_EventQueue *queue)
{
    queue->head = queue->tail = queue->pool ;
}
```

queue	a pointer to the queue which is to be reset.

Defining the State Model

A Mealy state model can be described by a transition matrix and a set of activities. We also keep the number of states and events in the model. These values are used for error checking and to allow us to perform two-dimensional array indexing from a one-dimensional transition matrix.

```
<<sync sm: interface data type declarations>>=
typedef struct {
    SSM_State initial_state ;
    uint8_t num_states ;
    uint8_t num_events ;
    SSM_State const *transitions ;
    SSM_Activity const *activities ;
    char const *const *event_names ;
    char const *const *state_names ;
} SSM_StateModel ;
```

State models can be used with multiple state machines to yield a set of identically behaving instances.

It is frequently the case that an execution sequence is started by a single event and we wish to signal the event and dispatch it immediately. We provide a convenience function for this purpose.

SSM_signalAndRun

```
<<sync sm: interface external declarations>>=
extern bool
SSM_signalAndRun(
    SSM_Machine *const machine,
    SSM_Event event) ;
```

machine	a pointer to a state machine which is to be signaled.
event	the event to signal to `machine`.

The `SSM_signalAndRun` function signals `event` to `machine` and immediately dispatches the event. Returns `true` if the dispatch happened correctly and `false` otherwise. A return value of `false` implies that at least part of the sequence of transitions and activity executions did *not* take place.

```
<<sync sm: external functions>>=
bool
SSM_signalAndRun(
    SSM_Machine *const machine,
```

```
    SSM_Event event)
{
    bool transitioned = false ;

    bool signaled = SSM_signal(machine, event) ;
    assert(signaled) ;

    if (signaled) {
        SSM_DispatchContext *const context = machine->context ;

        transitioned = context != NULL ?
                SSM_runContext(context) : SSM_dispatch(machine) ;
        assert(transitioned) ;
    }

    return transitioned ;
}
```

Defining the Execution Context

We need to allow for the situation where multiple state machines interact. In addition to a state machine signaling itself to drive its transitions, it may also need to signal another state machine as part of a larger execution sequencing scheme. To support the interactions of multiple state machines, we must make sure the ordering of event dispatch is correct and complete. The use of a context is optional. If the state machine involved does not signal other state machines, then the context is unnecessary.

```
<<sync sm: interface data type declarations>>=
typedef struct {
    struct ssm_machine *pool[SSM_DEFAULT_CONTEXT_SIZE] ;
    struct ssm_machine **head ;
    struct ssm_machine **tail ;
    struct ssm_machine **end ;
} SSM_DispatchContext ;
```

Contexts are usually quite small and we provide a reasonable default. The context size is the maximum number of state machine interactions that can happen when a single event is dispatched. For example, dispatching an event to one machine may cause it to signal another machine, and so on. The number of non-self directed events signaled to machines during this process determines the context size.

```
<<sync sm: interface macros>>=
#ifndef SSM_DEFAULT_CONTEXT_SIZE
#   define SSM_DEFAULT_CONTEXT_SIZE     4
#endif /* SSM_DEFAULT_CONTEXT_SIZE */
```

Note that the dispatch context is operated as yet another circular queue in an array.

Dispatch Context Operations

When an event is signaled to a state machine, that state machine is recorded within any associated dispatch context. The signaling operation is actually two parts. The first part places the event into the event queue of the state machine. This does *not* cause the event to be dispatched. The second part records the signal with the context so that when the context is eventually executed, the dispatch happens.

Note this is a emergent operation in the sense that as state machines transition and execute activities, those activities may, in turn, signal additional events to other state machines. That signaling must also be registered so it can be dispatched in proper order.

Adding a state machine to a dispatch context is the first required operation. This operation is used when a state machine is signaled so it can add itself to the sequence of dispatches to be executed.

SSM_addToContext

Return `true` if `machine` was added to `context` and `false` otherwise.

```
<<sync sm: static functions>>=
static bool
SSM_addToContext(
    SSM_DispatchContext *const context,
    SSM_Machine *const machine)
{
    assert(context != NULL) ;

    SSM_Machine **next_tail = context->tail + 1 ;
    if (next_tail >= context->end) {
        next_tail = context->pool ;
    }
    if (next_tail == context->head) {
        return false ;
    }

    *context->tail = machine ;
    context->tail = next_tail ;
    return true ;
}
```

context	a pointer to a state machine context to which `machine` is added.
machine	a pointer to a state machine which is to be added to `context`.

The other essential operation on dispatch contexts is to sequence through the state machines and dispatch any events.

SSM_runContext

```
<<sync sm: static functions>>=
static bool
SSM_runContext(
    SSM_DispatchContext *const context)
{
    assert(context != NULL) ;

    bool transitioned = false ;

    for (SSM_Machine **next = context->head ; next != context->tail ;
            next = context->head) {
        SSM_Machine *sm = *next ;

        context->head += 1 ;                              // ❶
        if (context->head >= context->end) {
            context->head = context->pool ;
        }

        transitioned = SSM_dispatch(sm) ;
        if (!transitioned) {
            break ;
        }
    }

    return transitioned ;
}
```

❶ It is possible that the invocation of `SSM_dispatch()` will add state machines to the context. This would happen if the state activity execution by a transition signaled a non-self directed event. Adding to the context only writes to the `tail` pointer, but diagnosing overflow does examine the value of the `head` pointer. Since we have consumed the current head value, we adjust the head pointer *before* invoking `SSM_dispatch()` to allow that slot in the context queue to be used. We use one slot in the context queue to diagnose overflow, we don't want to make it two.

context — a pointer to a state machine context for which a thread of control is dispatched.

Dispatch all the events associated with `context`. Returns `true` if the dispatch happened correctly and `false` otherwise. A return value of `false` implies that at least part of the sequence of transitions and activity executions did *not* take place.

Defining the State Machine

A state machine is then the combination of a variable to hold the current state, the event queue which holds events to be dispatched, a reference to the state model describing the dispatch behavior, and a reference to the dispatch context used for interacting with other state machines.

```
<<sync sm: interface data type declarations>>=
typedef struct ssm_machine {
    SSM_State current_state ;
    bool log_transitions ;
    SSM_EventQueue event_queue ;
    SSM_StateModel const *const state_model ;
    SSM_DispatchContext *context ;
    char const *name ;
} SSM_Machine ;
```

State Machine Operations

Several external operations are provided for a state machine. The primary operation is to signal an event to a state machine. The other important state machine operation is used inside of state activities to signal a self directed event.

The substantial difference between these two operations is whether the event is queued at the end of the event queue (normal state machine signaling) or at the beginning of the event queue (self-directed events which are dispatched first). We can factor out the code uses for the event queuing.

```
<<sync sm: static functions>>=
static bool
SSM_addToQueue(
    SSM_Machine *const machine,
    SSM_Event event,
    bool (*queue_func)(SSM_EventQueue *queue, SSM_Event event))
{
    assert(event < machine->state_model->num_events) ;
    if (event >= machine->state_model->num_events) {
        return false ;
    }

    bool inserted = queue_func(&machine->event_queue, event) ;
    assert(inserted) ;

    return inserted ;
}
```

SSM_signal

```
<<sync sm: interface external declarations>>=
extern bool
SSM_signal(
    SSM_Machine *const machine,
    SSM_Event event) ;
```

machine
: a pointer to a state machine which is to be signaled. *N.B.* `machine` must *not* be a pointer to the same state machine which is signaling the event (*i.e. not* `self`). This function adds the event to the end of the event queue for `machine` and this is inappropriate for a self-directed event.

event
: the event to signal to `machine`.

Return `true` if `event` was successfully added to the event queue for `machine` and `false` otherwise.

The implementation of `SSM_signal()` adds the event to the tail of the event queue and, if successful, adds the state machine to its context for later dispatch.

```
<<sync sm: external functions>>=
bool
SSM_signal(
    SSM_Machine *const machine,
    SSM_Event event)
{
    assert(machine != NULL) ;

    bool inserted = SSM_addToQueue(machine, event, SSM_insertAtTail) ;
    assert(inserted) ;
```

```
    if (inserted && machine->context != NULL) {
        inserted = SSM_addToContext(machine->context, machine) ;
    }

    return inserted ;
}
```

SSM_signalSelf

```
<<sync sm: interface external declarations>>=
extern bool
SSM_signalSelf(
    SSM_Machine *const machine,
    SSM_Event event) ;
```

machine
: a pointer to a state machine which is to be signaled. *N.B.* `machine` *must be* a pointer to the same state machine which is signaling the event (*i.e.* it must be `self`). This function adds the event to the beginning of the event queue for `machine` and this is necessary for a self-directed event.

event
: the event to signal to `machine`.

Return `true` if `event` was successfully added to the event queue for `machine` and `false` otherwise.

The implementation of `SSM_signalSelf()` is similar to `SSM_signal()` but does not add the state machine to the context. Once the dispatch of a state machine's events begins, the event queue for the machine is emptied.

```
<<sync sm: external functions>>=
bool
SSM_signalSelf(
    SSM_Machine *machine,
    SSM_Event event)
{
    assert(machine != NULL) ;

    bool inserted = SSM_addToQueue(machine, event, SSM_insertAtHead) ;
    assert(inserted) ;

    return inserted ;
}
```

SSM_resetMachine

```
<<sync sm: interface external declarations>>=
extern void
SSM_resetMachine(
    SSM_Machine *const machine) ;
```

machine
: a pointer to a state machine which is to be reset.

```
<<sync sm: external functions>>=
void
SSM_resetMachine(
    SSM_Machine *const machine)
{
```

```
    assert(machine != NULL) ;

    machine->current_state = machine->state_model->initial_state ;
    SSM_initEventQueue(&machine->event_queue) ;
}
```

The dispatch function operates as expected. Events are removed one at a time from the event queue of a state machine. The current state and event are used to compute a new state. We have the usual non-transitioning pseudo-states of ignore and can't happen. Assuming neither of those cases apply, the activity of the new state is obtained from the activity table and is executed. This continues until the event queue is empty.

SSM_dispatch

Dispatches all the events queued to `machine`.

```
<<sync sm: forward references>>=
static bool SSM_dispatch(
    SSM_Machine *const machine) ;
```

machine	a pointer to a state machine whose events are dispatched.

Implementation

```
<<sync sm: static functions>>=
static bool
SSM_dispatch(
    SSM_Machine *const machine)
{
    assert(machine != NULL) ;

    SSM_StateModel const *const state_model = machine->state_model ;
    bool transitioned = true ;

    for (int event = SSM_removeFromHead(&machine->event_queue) ; event >= 0 ;
            event = SSM_removeFromHead(&machine->event_queue)) {
        assert(event < state_model->num_events) ;
        SSM_State new_state = state_model->transitions[
                machine->current_state * state_model->num_events + event] ;        // ❶

        assert(new_state < state_model->num_states) ;
        if (new_state >= state_model->num_states) {
            transitioned = false ;
            break ;
        }

        if (machine->log_transitions) {
            SSM_logTransition(machine, event, new_state) ;
        }

        assert(new_state != SSM_State_CH) ;
        if (new_state == SSM_State_CH) {
            transitioned = false ;
            break ;
        } else if (new_state != SSM_State_IG) {
            machine->current_state = new_state ;
            SSM_Activity activity = state_model->activities[new_state] ;
            if (activity) {
                activity(machine) ;
            }
        }  // else the event is ignored
```

```
    }

    return transitioned ;
}
```

❶ Simulate the 2-D transition matrix using a 1-D data structure.

Event dispatch tracing can be logged for debugging purposes.

SSM_logEnable

```
<<sync sm: interface external declarations>>=
extern bool
SSM_logEnable(
    SSM_Machine *const machine,
    bool enable) ;
```

machine
a pointer to a state machine whose logging is to be contolled.

enable
a boolean which determines if the event dispatches of `machine` are to be logged.

The `SSM_logEnable` function enables or disables event dispatch logging according to the value of `enable` (`true` enables and `false` disables). Returns the previous value of the logging enable setting.

```
<<sync sm: external functions>>=
bool
SSM_logEnable(
    SSM_Machine *const machine,
    bool enable)
{
    bool previous_enable = machine->log_transitions ;
    machine->log_transitions = enable ;

    return previous_enable ;
}
```

SSM_logTransition

```
<<sync sm: forward references>>=
static void
SSM_logTransition(
    SSM_Machine *machine,
    SSM_Event event,
    SSM_State new_state) ;
```

```
<<sync sm: static functions>>=
static void
SSM_logTransition(
    SSM_Machine *machine,
    SSM_Event event,
    SSM_State new_state)
{
    assert(machine != NULL) ;

    SSM_StateModel const *const state_model = machine->state_model ;
```

```
    assert(state_model != NULL) ;
    assert(event < state_model->num_events) ;
    assert(new_state < state_model->num_states) ;

    char const *new_state_name ;

    switch (new_state) {
    case SSM_State_IG:
        new_state_name = "IG" ;
        break ;

    case SSM_State_CH:
        new_state_name = "CH" ;
        break ;

    default:
        new_state_name = state_model->state_names[new_state] ;
        break ;
    }

    printf("%s <- %s: %s -> %s\n",
            machine->name,
            state_model->event_names[event],
            state_model->state_names[machine->current_state],
            new_state_name) ;
}
```

State Model Definition Macros

Implementing a state model requires defining both the code for the state activities and the necessary initialized data structures to drive the operations. This can be tedious. To save some typing, we define a set of macros to help create the variables and initializers required for a state machine.

As an example, we define a state machine to be used in testing. We start by defining enumerations for the states and events. We also supply names for the events that can be useful in debugging.

```
<<sync sm: test data definitions>>=
typedef enum {
    tms_IDLE,
    tms_STARTING,
    tms_RUNNING,
    tms_PAUSED,

    tms_STATE_COUNT          // last
} TestModelStates ;
```

It is convenient to define a trailing enumerator to use as a count since we need to provide that value later.

```
<<sync sm: interface macros>>=
#   define  SSM_DEFINE_STATE_NAMES(model, nstates)      \
    char const *const model ## __SNAME[nstates]
```

```
<<sync sm: interface macros>>=
#   define  SSM_NAME_STATE(snum, name)                  \
    [snum] = #name
```

```
<<sync sm: test data definitions>>=
SSM_DEFINE_STATE_NAMES(test_model, tms_STATE_COUNT) = {
    SSM_NAME_STATE(tms_IDLE, IDLE),
```

```
    SSM_NAME_STATE(tms_STARTING, STARTING),
    SSM_NAME_STATE(tms_RUNNING, RUNNING),
    SSM_NAME_STATE(tms_PAUSED, PAUSED),
} ;
```

```
<<sync sm: test data definitions>>=
typedef enum {
    tme_RUN,
    tme_PAUSE,
    tme_STOP,
    tme_CONT,

    tme_EVENT_COUNT         // last
} TestModelEvents ;
```

```
<<sync sm: interface macros>>=
#   define  SSM_DEFINE_EVENT_NAMES(model, nevents)      \
    char const *const model ## __ENAME[nevents]
```

```
<<sync sm: interface macros>>=
#   define  SSM_NAME_EVENT(enum, name)                  \
    [enum] = #name
```

```
<<sync sm: test data definitions>>=
SSM_DEFINE_EVENT_NAMES(test_model, tme_EVENT_COUNT) = {
    SSM_NAME_EVENT(tme_RUN, RUN),
    SSM_NAME_EVENT(tme_PAUSE, PAUSE),
    SSM_NAME_EVENT(tme_STOP, STOP),
    SSM_NAME_EVENT(tme_CONT, CONT),
} ;
```

The next step is to define the activity table. This table is an array of *states* elements which are the pointers to the activity functions for the state model. We provide some helper macros to enforce a naming convention which is used to accumulate the various data structures required to define a state model.

```
<<sync sm: interface macros>>=
#define SSM_DEFINE_ACTIVITY_TABLE(model, nstates)           \
SSM_Activity const model ## __ACT[nstates]
```

```
<<sync sm: interface macros>>=
#define SSM_DEFINE_ACTIVITY(state, func)                    \
[state] = func
```

For our test case example, the activity table appears as:

```
<<sync sm: test data definitions>>=
SSM_DEFINE_ACTIVITY_TABLE(test_model, tms_STATE_COUNT) = {
    SSM_DEFINE_ACTIVITY(tms_IDLE, idle_activity),
    SSM_DEFINE_ACTIVITY(tms_STARTING, starting_activity),
    SSM_DEFINE_ACTIVITY(tms_RUNNING, running_activity),
    SSM_DEFINE_ACTIVITY(tms_PAUSED, NULL),
} ;
```

Note you may specify an activity function as `NULL`. This is used for the case of an empty activity and the event dispatch will not attempt to invoke any activity upon the transition.

The next job is to define the transition matrix. The transition matrix is an array *states* by *events* in size.

The following macro is used to begin the definition of the transition matrix.

```
<<sync sm: interface macros>>=
#define SSM_DEFINE_TRANS_TABLE(model, nstates, nevents)      \
SSM_State const model ## __TRAN[nstates][nevents]
```

Then for each cell in the matrix, the following macro is used to specify the transition. The macro specifies to which new state a given event causes a transition from a given current state.

```
<<sync sm: interface macros>>=
#define SSM_DEFINE_TRANSITION(current, event, new)           \
[current][event] = new
```

Our test model has the following transition table.

Table 17.1: Test State Transition Matrix

	RUN	**PAUSE**	**STOP**	**CONT**
IDLE	STARTING	CH	CH	CH
STARTING	CH	CH	CH	RUNNING
RUNNING	IG	PAUSED	IDLE	CH
PAUSED	RUNNING	PAUSED	IDLE	CH

The event names are listed across the top row of the table. The state names are listed as the first column of each row.

For the test model, the transition matrix can be encoded as follows.

```
<<sync sm: test data definitions>>=
SSM_DEFINE_TRANS_TABLE(test_model, tms_STATE_COUNT, tme_EVENT_COUNT) = {
    SSM_DEFINE_TRANSITION(tms_IDLE, tme_RUN, tms_STARTING),
    SSM_DEFINE_TRANSITION(tms_IDLE, tme_PAUSE, SSM_State_CH),
    SSM_DEFINE_TRANSITION(tms_IDLE, tme_STOP, SSM_State_CH),
    SSM_DEFINE_TRANSITION(tms_IDLE, tme_CONT, SSM_State_CH),

    SSM_DEFINE_TRANSITION(tms_STARTING, tme_RUN, SSM_State_CH),
    SSM_DEFINE_TRANSITION(tms_STARTING, tme_PAUSE, SSM_State_CH),
    SSM_DEFINE_TRANSITION(tms_STARTING, tme_STOP, SSM_State_CH),
    SSM_DEFINE_TRANSITION(tms_STARTING, tme_CONT, tms_RUNNING),

    SSM_DEFINE_TRANSITION(tms_RUNNING, tme_RUN, SSM_State_IG),
    SSM_DEFINE_TRANSITION(tms_RUNNING, tme_PAUSE, tms_PAUSED),
    SSM_DEFINE_TRANSITION(tms_RUNNING, tme_STOP, tms_IDLE),
    SSM_DEFINE_TRANSITION(tms_RUNNING, tme_CONT, SSM_State_CH),

    SSM_DEFINE_TRANSITION(tms_PAUSED, tme_RUN, tms_RUNNING),
    SSM_DEFINE_TRANSITION(tms_PAUSED, tme_PAUSE, tms_PAUSED),
    SSM_DEFINE_TRANSITION(tms_PAUSED, tme_STOP, tms_IDLE),
    SSM_DEFINE_TRANSITION(tms_PAUSED, tme_CONT, SSM_State_CH),
} ;
```

Note that every one of the 16 elements of the transition matrix must be specified. Also note the use `SSM_State_IG` and `SSM_State_CH` as transition matrix elements.

Before defining any state models, we need a context in which they run. For a state machine which does not signal any other state machine, a context is not needed and can be specified as `NULL` when the state model is defined.

```
<<sync sm: interface macros>>=
#define SSM_DEFINE_CONTEXT(var)                              \
SSM_DispatchContext var = {                                  \
```

```
    .pool = {0},                                                \
    .head = var.pool,                                           \
    .tail = var.pool,                                           \
    .end = var.pool + sizeof(var.pool) / sizeof(var.pool[0])    \
}
```

```
<<sync sm: interface macros>>=
#define SSM_CONTEXT_INITIALIZER(var, member)                    \
.member = {                                                     \
    .pool = {0},                                                \
    .head = var.member.pool,                                    \
    .tail = var.member.pool,                                    \
    .end = var.member.pool +                                    \
        sizeof(var.member.pool) / sizeof(var.member.pool[0])    \
}
```

The state model is a synthesis of the transition matrix and the activities table.

```
<<sync sm: interface macros>>=
#   define SSM_DEFINE_STATE_MODEL(model, initial, nstates, nevents) \
SSM_StateModel const model = {                                      \
    .initial_state = initial,                                       \
    .num_states = nstates,                                          \
    .num_events = nevents,                                          \
    .transitions = (SSM_State const *)model ## __TRAN,              \
    .activities = model ## __ACT,                                   \
    .event_names = model ## __ENAME,                                \
    .state_names = model ## __SNAME,                                \
} /* ❶ */
```

❶ *N.B.* the cast for the `.transitions` member is a "cheat". We have defined the transition matrix as a *states* by *events* two dimensional matrix. Of course, the number of states and events varies for each state model. But we "know" that "C" will layout the memory in row order first. So, we cast the two dimensional matrix to be a one dimensional array and perform the two dimensional indexing explicitly in code (this is one reason we need knowledge of the number of events for the state model). This allows us to use the same generic code irrespective of the number of states and events in a given state model.

Note the state model is `const` data and can be used to control multiple state machines.

For our test model, the definition appears as:

```
<<sync sm: test data definitions>>=
SSM_DEFINE_STATE_MODEL(test_model, tms_IDLE, tms_STATE_COUNT, tme_EVENT_COUNT) ;
```

The intended use case for a state machine is to embed it into a surrounding structure which contains other elements. This gives the state machine its access to any required data.

For the test model, we define the following structure to enclose it.

```
<<sync sm: test data definitions>>=
typedef struct {
    SSM_Machine machine ;
    unsigned count ;
} CountingMachine ;
```

N.B. it is *most convenient* to place the state machine as the **first** member of the enclosing structure. This allows us to treat a pointer to a `CountingMachine` as a pointer to a `SSM_Machine` and the reference given to the activity functions can have its type recovered most conveniently.

For defining a state machine enclosed in a structure, the helper macro can be used to provide the initializer for a member of the enclosing structure.

```
<<sync sm: interface macros>>=
#define SSM_DEFINE_MACHINE(var, model, initial, cntx)    \
SSM_Machine var = {                                      \
    .current_state = initial,                            \
    .log_transitions = false,                            \
    .event_queue = {                                     \
        .pool = {0},                                     \
        .head = var.event_queue.pool,                    \
        .tail = var.event_queue.pool,                    \
        .end = var.event_queue.pool +                    \
            sizeof(SSM_EventPool) / sizeof(SSM_Event)    \
    },                                                   \
    .state_model = model,                                \
    .context = cntx,                                     \
    .name = #var,                                        \
}
```

```
<<sync sm: interface macros>>=
#define SSM_MACHINE_INITIALIZER(var, member, model, initial, cntx) \
.member = {                                              \
    .current_state = initial,                            \
    .log_transitions = false,                            \
    .event_queue = {                                     \
        .pool = {0},                                     \
        .head = var.member.event_queue.pool,             \
        .tail = var.member.event_queue.pool,             \
        .end = var.member.event_queue.pool +             \
            sizeof(SSM_EventPool) / sizeof(SSM_Event)    \
    },                                                   \
    .state_model = model,                                \
    .context = cntx,                                     \
    .name = #var "-" #member                             \
}
```

var
: The name of the variable holding the state machine. The variable is assumed to be of some structure type.

member
: The name of the member of the enclosing structure holding the state machine.

model
: The name of the state model which defines the behavior.

initial
: The initial value of the state machine's current state.

cntx
: The machine execution context, or `NULL` for isolated state machines.

For our test machine, this becomes:

```
<<sync sm: test data definitions>>=
CountingMachine test_machine = {
    SSM_MACHINE_INITIALIZER(test_machine, machine, &test_model, tms_IDLE, NULL),
    .count = 0,
} ;
```

Macro Summary

In summary, the following sequence can be used to define the necessary data structures using the helper macros.

- Define an enumeration for the state model events. The enumerators must start at zero and be sequential. Including a last enumerator as a count is convenient. Define an enumeration for the states also.
- Define the state and events names using the `SSM_DEFINE_STATE_NAMES` / `SSM_NAME_STATE` and `SSM_DEFINE_EVENT_NAM` / `SSM_NAME_EVENT` macros.
- Define the activity table using the `SSM_DEFINE_ACTIVITY_TABLE` and `SSM_DEFINE_ACTIVITY` macros.
- Define the transition matrix using the `SSM_DEFINE_TRANS_TABLE` and `SSM_DEFINE_TRANSITION` macros.
- Define the state model using the `SSM_DEFINE_STATE_MODEL` macro.
- If a dispatch context is needed, the `SSM_DEFINE_CONTEXT` can be used to define a variable. If the context is included in a larger structure, the `SSM_CONTEXT_INITIALIZER` macro is used to define a designated initializer for the context.
- Declare a data structure to contain the state machine and any application specific data.
- Initialize any state machine instances using the `SSM_DEFINE_MACHINE` to define a stand alone variable. If the state machine is parter of a larger structure, use the `SSM_MACHINE_INITIALIZER` macro to initialize the state machine member.
- Place all state machines which interact into the same dispatch context.
- Use the same state model for multiple state machines which are to behave the same.

Code Layout

In literate programming terminology, a *chunk* is a named part of the final program. The program chunks form a tree and the root of that tree is named * by default. We follow the convention of naming the root the same as the output file name. The process of extracting the program tree formed by the chunks is called *tangle*. By the default the program, **atangle**, extracts the root chunk to produce the “C” source file.

Header file

```
<<sync_sm.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Farfalle
 *
 * Module:
 *   Synchronous State Machine Dispatch
 *--
 */
#ifndef SYNC_SM_H_
#define SYNC_SM_H_

/*
 * Include files
 */
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
<<sync sm: interface include files>>
/*
```

```
 * Constants
 */
<<sync sm: interface constants>>
/*
 * Macros
 */
<<sync sm: interface macros>>
/*
 * Forward References
 */
<<sync sm: interface forward references>>
/*
 * Data Type Declarations
 */
<<sync sm: interface data type declarations>>
/*
 * External Declarations
 */
<<sync sm: interface external declarations>>
/*
 * Inline Function Definitions
 */
<<sync sm: interface inline functions>>
#endif /* SYNC_SM_H_ */
```

Source file

```
<<sync_sm.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Farfalle
 *
 * Module:
 *   Synchronous State Machine Dispatch
 *
 * Conditional Compilation:
 * The following pre-processor symbols may be redefine
 *
 * SSM_DEFAULT_EVENT_POOL_SIZE ==> the size of the event queue storage (default 4)
 * SSM_DEFAULT_CONTEXT_SIZE ==> the size of the context queue storage (default 4)
 *--
 */

/*
 * Include files
 */
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>

<<sync sm: include files>>
#include "sync_sm.h"
/*
 * Constants
 */
<<sync sm: constants>>
```

```
/*
 * Data Type Declarations
 */
<<sync sm: data type declarations>>
/*
 * External Declarations
 */
<<sync sm: external declarations>>
/*
 * Forward References
 */
<<sync sm: forward references>>
/*
 * Static Inline Functions
 */
<<sync sm: static inline functions>>
/*
 * External Inline Functions
 */
<<sync sm: external inline functions>>
/*
 * Static Data
 */
<<sync sm: static data>>
/*
 * Static Functions
 */
<<sync sm: static functions>>
/*
 * External Functions
 */
<<sync sm: external functions>>
```

Unit testing

To test the state machine functions, we use the Unity testing framework. The Unity framework is small, simple, pure "C", and can be used in low resource contexts. To use the Unity framework, we need the header file.

```
<<sync sm: test include files>>=
#include "unity.h"
```

The Unity testing framework has the usual concepts:

- The entire test suite consists of running a set of test groups.
- Test groups contain a set of test cases.
- Each test group has a setup and tear down run before the test case is executed.
- A test invokes code and has available a set of assertion macros that are used to record the result.
- The framework orchestrates the running of the test suite and reporting the results.

Specifying State Activities

The state machine for this test were defined in the previous section as an example of how to use the helper macros. Here we complete the specification by giving the code associated with the state activities.

```
<<sync sm: test forward references>>=
static void idle_activity(SSM_Machine *const machine) ;
static void starting_activity(SSM_Machine *const machine) ;
static void running_activity(SSM_Machine *const machine) ;
```

```
<<sync sm: test static functions>>=
static void
idle_activity(
    SSM_Machine *const machine)
{
    CountingMachine *const self = CONTAINER_OF(machine, CountingMachine, machine) ;
    self->count++ ;
}
static void
starting_activity(
    SSM_Machine *const machine)
{
    CountingMachine *const self = CONTAINER_OF(machine, CountingMachine, machine) ;
    self->count++ ;
    SSM_signalSelf(machine, tme_CONT) ;
}
static void
running_activity(
    SSM_Machine *const machine)
{
    CountingMachine *const self = CONTAINER_OF(machine, CountingMachine, machine) ;
    self->count++ ;
}
```

Testing insertion

```
<<sync sm: test include files>>=
#include "sync_sm.h"
```

The setup and tear down have to be specified, even if they are empty.

```
<<sync sm: test groups>>=
void
setUp(void)
{
}

void
tearDown(void)
{
}
```

In the Unity framework, a test group uses a test group runner to execute the individual tests in the group.

```
<<sync sm: test runners>>=
static void
run_dispatch_tests(void)
{
    RUN_TEST(run_event) ;
    RUN_TEST(pause_event) ;
    RUN_TEST(pause_event_2) ;
    RUN_TEST(stop_event) ;
}
```

```
<<sync sm: test groups>>=
static void
run_event(void)
{
    TEST_ASSERT_EQUAL_UINT(tms_IDLE, test_machine.machine.current_state) ;
    TEST_ASSERT_EQUAL_UINT(0, test_machine.count) ;

    SSM_signalAndRun(&test_machine.machine, tme_RUN) ;

    TEST_ASSERT_EQUAL_UINT(tms_RUNNING, test_machine.machine.current_state) ;
    TEST_ASSERT_EQUAL_UINT(2, test_machine.count) ;
}
```

```
<<sync sm: test groups>>=
static void
pause_event(void)
{
    SSM_signalAndRun(&test_machine.machine, tme_PAUSE) ;

    TEST_ASSERT_EQUAL_UINT(tms_PAUSED, test_machine.machine.current_state) ;
    TEST_ASSERT_EQUAL_UINT(2, test_machine.count) ;
}
```

```
<<sync sm: test groups>>=
static void
pause_event_2(void)
{
    SSM_signalAndRun(&test_machine.machine, tme_PAUSE) ;

    TEST_ASSERT_EQUAL_UINT(tms_PAUSED, test_machine.machine.current_state) ;
    TEST_ASSERT_EQUAL_UINT(2, test_machine.count) ;
}
```

```
<<sync sm: test groups>>=
static void
stop_event(void)
{
    SSM_signalAndRun(&test_machine.machine, tme_STOP) ;

    TEST_ASSERT_EQUAL_UINT(tms_IDLE, test_machine.machine.current_state) ;
    TEST_ASSERT_EQUAL_UINT(3, test_machine.count) ;
}
```

Running the tests

The main entry point of the framework is handed the function to run all the test groups.

```
<<sync sm: test main function>>=
int
main(
    int argc,
    const char **argv)
{
    UNITY_BEGIN() ;

    run_dispatch_tests() ;
    SSM_resetMachine(&test_machine.machine) ;
    test_machine.count = 0 ;
```

```
    run_dispatch_tests() ;

    return UNITY_END() ;
}
```

Testing Source

```
<<sync_sm_test.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Farfalle
 *
 * Module:
 *   Source code for Synchronous State Model Dispatch tests
 *--
 */

/*
 * Include files
 */
#include <stdio.h>
#include <string.h>
<<sync sm: test include files>>
/*
 * Constants
 */
<<sync sm: test constants>>
/*
 * Data Type Declarations
 */
<<sync sm: test type declarations>>
/*
 * Forward references
 */
<<sync sm: test forward references>>
/*
 * Data Definitions
 */
<<sync sm: test data definitions>>
/*
 * Static functions
 */
<<sync sm: test static functions>>
/*
 * Test groups
 */
<<sync sm: test groups>>
/*
 * Test runners
 */
<<sync sm: test runners>>
/*
 * Test collections
 */
<<sync sm: all tests runner>>
/*
 * Test main function
```

```
 */
<<sync sm: test main function>>
```

Chapter 18

Bipartite Buffer

A bipartite buffer is a circular queue implemented in an array with the characteristic that the allocated space is contiguous. Typically, circular queues implemented in arrays must handle the modulus arithmetic to wrap around from the end of an array to its beginning. The bipartite buffer makes contiguous allocations by splitting the array into two parts when there is insufficient space at the end of the array to satisfy a request for space.

In this section, we develop a set of functions to implement a bipartite buffer. This implementation also borrows concepts from lock-free implementations in Rust. See also.

Requirements

This implementation differs from those referenced previously. The specific requirements are:

- Allocated blocks form a FIFO queue.
- Allocation requests are sized and that size does not change while an allocated slot is in use.
- No attempt is made to insure that the code is safe under preemption. Users must take what precautions are necessary for the context in which the buffer is used.
- To limit copying of data on the writing side, allocations are made in two steps.
  - The buffer is *probed* to determine if the requested space is available. Probing will return a pointer to the allocated slot.
  - Data can them be written to the probed area.
  - After the buffer slot is filled, it is *pushed*. Pushing makes the buffer slot and its data visible to any reader.
- Reading is accomplished by *peeking* to determine if a slot is available. Peeking returns a pointer to the slot at the head of queue. Once the data is handled by the reader, the buffer is *popped*, returning the slot at the head of the read side of the queue back to the buffer.
- The sequence of probe then push is enforced to occur in pairs on the write side of the queue.
- The peek operation is idempotent and the state of the read side of the queue is only modified by the pop operation.

Bipartite Buffer Operations

This section describes the operations supported by the bipartite buffer.

bip_probe

```
<<bip_buffer: external function declarations>>=
void *
bip_probe(
    BipBuffer *const bipbuf,
    size_t probe_size) ;
```

bipbuf
A pointer to a bip buffer from which a slot is to be allocated.

probe_size
The number of bytes to allocate to the slot.

The `bip_probe` function attempts to allocate a buffer slot of `probe_size` bytes from the buffer pointed to by `bipbuf`. If successful, the return value is a pointer to a slot of at least `probe_size` contiguous bytes in memory whose address is aligned the same as a variable of `size_t` type. The return value is `NULL` if the probe fails because of lack of space or probing was not an appropriate operation for the state of the buffer.

After invoking `bip_probe`, data may be written to the buffer using the returned pointer. After writing, `bip_push` must be invoked to make the data in the allocated slot available to the queue reader. Each `bip_probe` must be followed by `bip_push` except in the case where a `bip_probe` invocation failed. In the failed case, it is allowed to `bip_probe` consecutively since more space may have become available as data is read out of the buffer.

The implementation of `bip_probe` is distinctive since it uses a state machine to control the bip buffer. These state machines are discussed in a following section.

```
<<bip_buffer: external function definitions>>=
void *
bip_probe(
    BipBuffer *const bipbuf,
    size_t probe_size)
{
    rtcheck_not_zero_return(probe_size, NULL) ;

    bipbuf->probe_size = probe_size ;
    bool ran = SSM_signalAndRun(&bipbuf->probe_sm, prbe_Probe) ;
    assert(ran) ;

    return ran ? bipbuf->probe : NULL ;
}
```

bip_push

```
<<bip_buffer: external function declarations>>=
bool
bip_push(
    BipBuffer *const bipbuf) ;
```

bipbuf
A pointer to a bip buffer for which the last slot allocated by `bip_probe` is to be made available to be read from the queue.

The `bip_push` function makes the buffer from the previous invocation of `bip_probe` available to be read from the queue. The return value is `true` if the push operation succeeded and `false` otherwise.

Because a state machine runs the buffer operations, the implementation of `bip_push` is brief.

```
<<bip_buffer: external function definitions>>=
bool
bip_push(
    BipBuffer *const bipbuf)
{
    bool ran = SSM_signalAndRun(&bipbuf->probe_sm, prbe_Push) ;
    assert(ran) ;
    return ran ;
}
```

bip_peek

```
<<bip_buffer: external function declarations>>=
void *
bip_peek(
    BipBuffer *const bipbuf,
    size_t *const len) ;
```

bipbuf
A pointer to a bip buffer from which a slot is to be read.

len
A pointer to memory object where the length of the slot is returned. If `len` is not `NULL`, and a slot can be read, the length of the slot is returned by reference. If no slot is available, then the object pointed to by `len` is not modified. If `len` is given as `NULL`, then no length information is returned.

The return value is a pointer to a queue slot which can be read. The return value is `NULL` if no slot is available, *i.e.* the queue is empty. The `bip_peek` function does *not* modify the internal state of the bip buffer and may be invoked by the queue ready as necessary to determine if data is available for reading.

```
<<bip_buffer: external function definitions>>=
void *
bip_peek(
    BipBuffer *const bipbuf,
    size_t *const len)
{
    void *peek = bipbuf->peek ;
    if (len != NULL) {
        if (peek != NULL) {
            *len = *(size_t *)((uintptr_t)peek - sizeof(size_t)) ;  // ❶
        } else {
            *len = 0 ;
        }
    }

    return peek ;
}
```

❶ The size of the slot is located one word before the `peek` pointer.

bip_pop

```
<<bip_buffer: external function declarations>>=
void *
bip_pop(
    BipBuffer *const bipbuf,
    size_t *len) ;
```

bipbuf
A pointer to a bip buffer from which a slot is to be removed.

len
A pointer to memory object where the length of the slot is returned.

The `bip_pop` function removes the slot at the head of the read side of the queue and returns it to the buffer for subsequent allocation. It then performs an implicit `bip_peek` operation. The return value of the function and the semantics of the `len` parameter are identical as for `bip_peek`. This behavior is intended to make iterating to empty the buffer more convenient.

```
<<bip_buffer: external function definitions>>=
void *
bip_pop(
    BipBuffer *const bipbuf,
    size_t *len)
{
    bool ran = SSM_signalAndRun(&bipbuf->pop_sm, pope_Pop) ;
    assert(ran) ;

    return ran ? bip_peek(bipbuf, len) : NULL ;
}
```

bip_reset

```
<<bip_buffer: external function declarations>>=
void
bip_reset(
    BipBuffer *const bipbuf) ;
```

bipbuf
A pointer to a bip buffer which is to be reset.

The `bip_reset` function discards all the data in the bip buffer and returns it to original, empty state.

```
<<bip_buffer: external function definitions>>=
void
bip_reset(
    BipBuffer *bipbuf)
{
    bipbuf->read = bipbuf->write = bipbuf->begin ;
    bipbuf->watermark = bipbuf->end ;
    bipbuf->peek = bipbuf->probe = NULL ;
    bipbuf->probe_size = 0 ;
    SSM_resetMachine(&bipbuf->probe_sm) ;
    SSM_resetMachine(&bipbuf->pop_sm) ;
}
```

Bipartite Buffer Data Structures

The following data structure is used to manage the bipartite buffer.

```
<<bip_buffer: interface data type declarations>>=
typedef struct {
    void *read ;
    void *write ;
    void *watermark ;
    void *const begin ;
    void *const end ;
    void *peek ;
    void *probe ;
    size_t probe_size ;
    SSM_DispatchContext cntx ;
    SSM_Machine probe_sm ;
    SSM_Machine pop_sm ;
} BipBuffer ;
```

read
: A pointer to the location in the buffer which the next read operation will affect.

write
: A pointer to the location in the buffer which the next write operation will affect.

watermark
: A pointer to the end of valid data in the buffer.

begin
: A pointer to the beginning of the array which holds the bip buffer.

end
: A pointer to one past the end of the array which holds the bip buffer.

peek
: A pointer to the location in the buffer which can be read as a result of performing a `peek` operation.

probe
: A pointer to the location in the buffer which was allocated by a `probe` operation.

probe_size
: The number of bytes for the next probe operation.

cntx
: The execution context for the state machines.

probe_sm
: The write or “probe” side state machine.

pop_sm
: The read or “pop” side state machine.

Defining a Bipartite Buffer

Since defining a bipartite buffer requires allocating storage for the buffer proper and for its controlling data structure, we fall back to our usual technique to use the pre-processor to save some typing and get all the details correct.

```
<<bip_buffer: macros>>=
#define     BIP_DEFINE_BUFFER(name, size)                   \
static alignas(max_align_t) char                            \
    name ## __POOL[((size) + sizeof(size_t) - 1) & ~(sizeof(size_t) - 1)] ; \
static BipBuffer name = {                                   \
    .read = name ## __POOL,                                 \
    .write = name ## __POOL,                                \
    .watermark = name ## __POOL + COUNTOF(name ## __POOL),  \
    .begin = name ## __POOL,                                \
    .end = name ## __POOL + COUNTOF(name ## __POOL),        \
    .probe = NULL,                                          \
    .peek = NULL,                                           \
    SSM_CONTEXT_INITIALIZER(name, cntx),                    \
    SSM_MACHINE_INITIALIZER(name, probe_sm, &bip_probe_model, prbs_Empty, &name.cntx),\
    SSM_MACHINE_INITIALIZER(name, pop_sm, &bip_pop_model, pops_Contiguous_Empty, &name. ↩
        cntx),\
}
```

name
: A variable name by which the bipartite buffer can be referenced.

size
: The number of bytes of storage allocated for the bipartite buffer.

The `BIP_DEFINE_BUFFER` macro defines the storage and control data for a bipartite buffer. The variables used for the buffer are defined to have file static scope.

State Models

This implementation is distinctive in that the control of the buffer is accomplished by a set of interacting state machines.

Probe State Model

The following diagram shows the state model for the write side, or “probe” side, of the bipartite buffer. The diagram uses a subset of UML state diagram graphical notation. The state model is of the Mealy type, *i.e.* state activities are associated with the state and are executed upon entry into the state. The state activities are shown in a high-level pseudo-code.

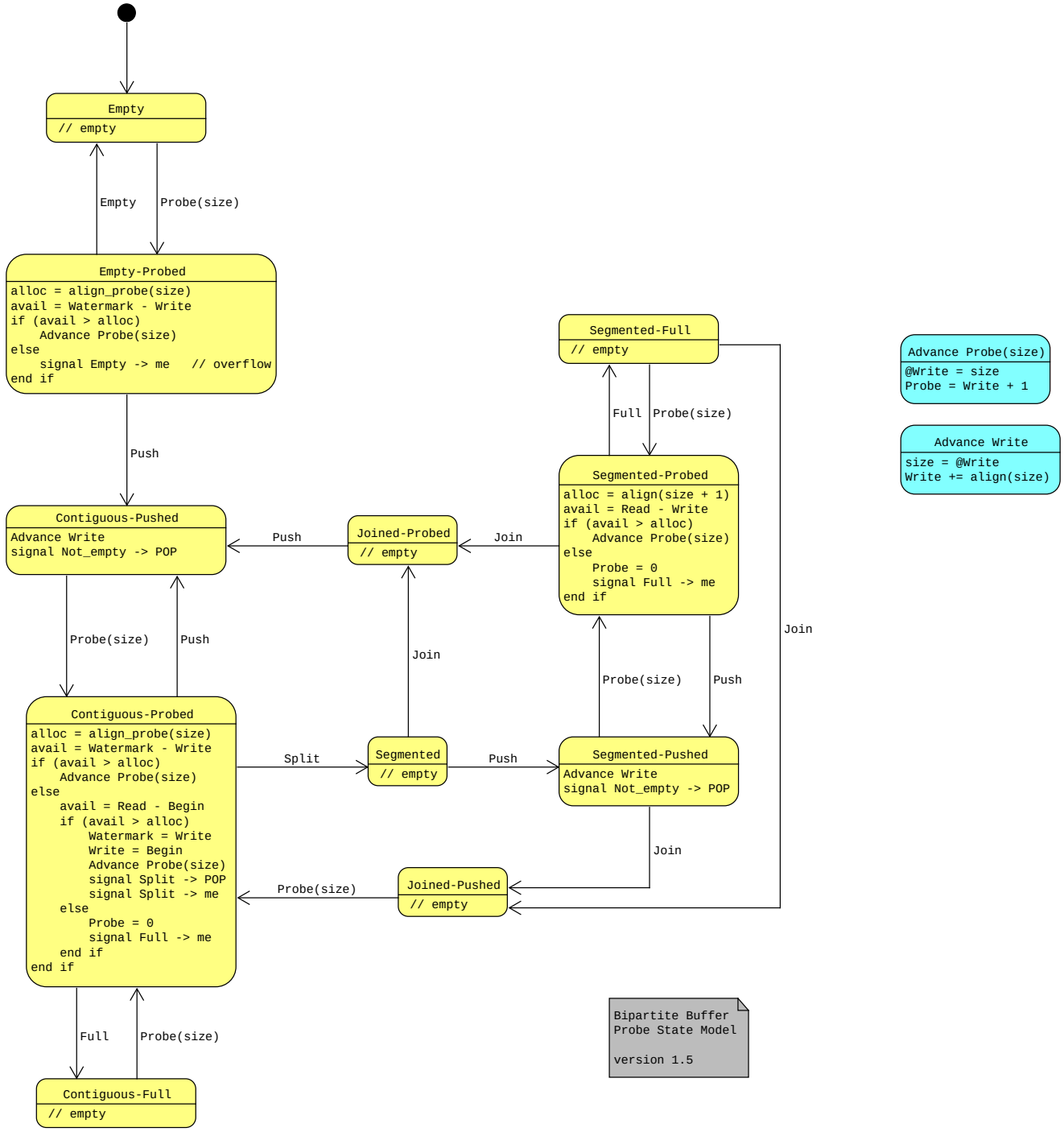

Figure 18.1: Bipartite Buffer Probe State Model

Overall, the states are divided into roughly two groups, those associated with allocating the buffer when it is still one contiguous block of memory and those associated with the buffer when it has been split into two blocks. It is the probe side which determines when the buffer is split into two parts (*cf.* to the read or pop side which determines when the buffer is joined back together). Since all successful probes of the buffer must be followed by a push operation, the allowed transitions in the state model control the sequencing of write side operations.

The buffer starts out in the **Empty** state. The only allowed operation is to probe the buffer asking for a slot to be allocated.

Probing causes the transition to the **Empty-Probed** state. The **Empty-Probed** state determines if there is sufficient room for the requested size. In the odd case where the initial probe request is for a size greater than the whole buffer, we fail and go back to being **Empty**. Otherwise, the size of the request is captured and a valid probe pointer is computed.

When the allocated slot is pushed, the **Contiguous-Pushed** state is entered. This state is responsible for advancing the write side pointer, which makes the newly pushed slot available to be read. It also signals the pop side state model that the buffer is no longer empty.

A subsequent probe of the buffer causes a transition to the **Contiguous-Probed** state. This state, like the **Empty-Probed** state primarily attempts to find space for the slot in the contiguous buffer. If space remains between the Watermark and Write pointers, then it is allocated just as in the **Empty-Probed** state. However, if the requested size cannot be allocated, the state activity attempts to *split* the buffer. If there is room for the allocation at the beginning of the buffer, then the buffer may be split. This would be the case if there have been any *pop* operations on the read side of the buffer which have the effect of making space available at the beginning of the buffer. If there is no room at the beginning, then the buffer is effectively full and the state model drives itself to the **Contiguous-Full** state. Assuming the split is possible, we set the Watermark value to the Write value. Once split, the Watermark now indicates the end of data in the buffer. The Write location is wrapped around to the beginning of the buffer and the pop state model is signaled to indicate the new condition of the buffer. When split, the **Contiguous-Probed** state drives itself to the **Segmented** state to indicate that the rules for probing and pushing are now working on a segmented buffer rather than a contiguous one.

The **Segmented-Pushed**, **Segmented-Probed**, and **Segmented-Full** states are counterparts to the similarly named **Contiguous-XXX** states. There are two differences in the logic when the buffer is segmented. First, the available space in the buffer is the number of bytes between the Read pointer and the Write pointer. Because the buffer is segmented, the Write pointer has "wrapped around" to the beginning of the buffer and the space available for allocation is between the Read and Write locations. Second, since we are already split, if there is insufficient space to meet a probe request, then the buffer is considered full and no decision about splitting is computed.

When the buffer is split, eventually the queue reader will "pop" enough data off the head of the queue that the pop state model will decide that the segmented buffer can be joined back to be a contiguous buffer. The pop state model signals the Join event to indicate this condition. The Join event causes a transition to either the **Joined-Probed** state or the **Joined-Pushed** state. Which state is entered is determined by which probe side operation has been most recently executed and therefore which probe side operation is allowed next.

Pop State Model

The following diagram shows the state model for the read side, or "pop" side, of the bipartite buffer.

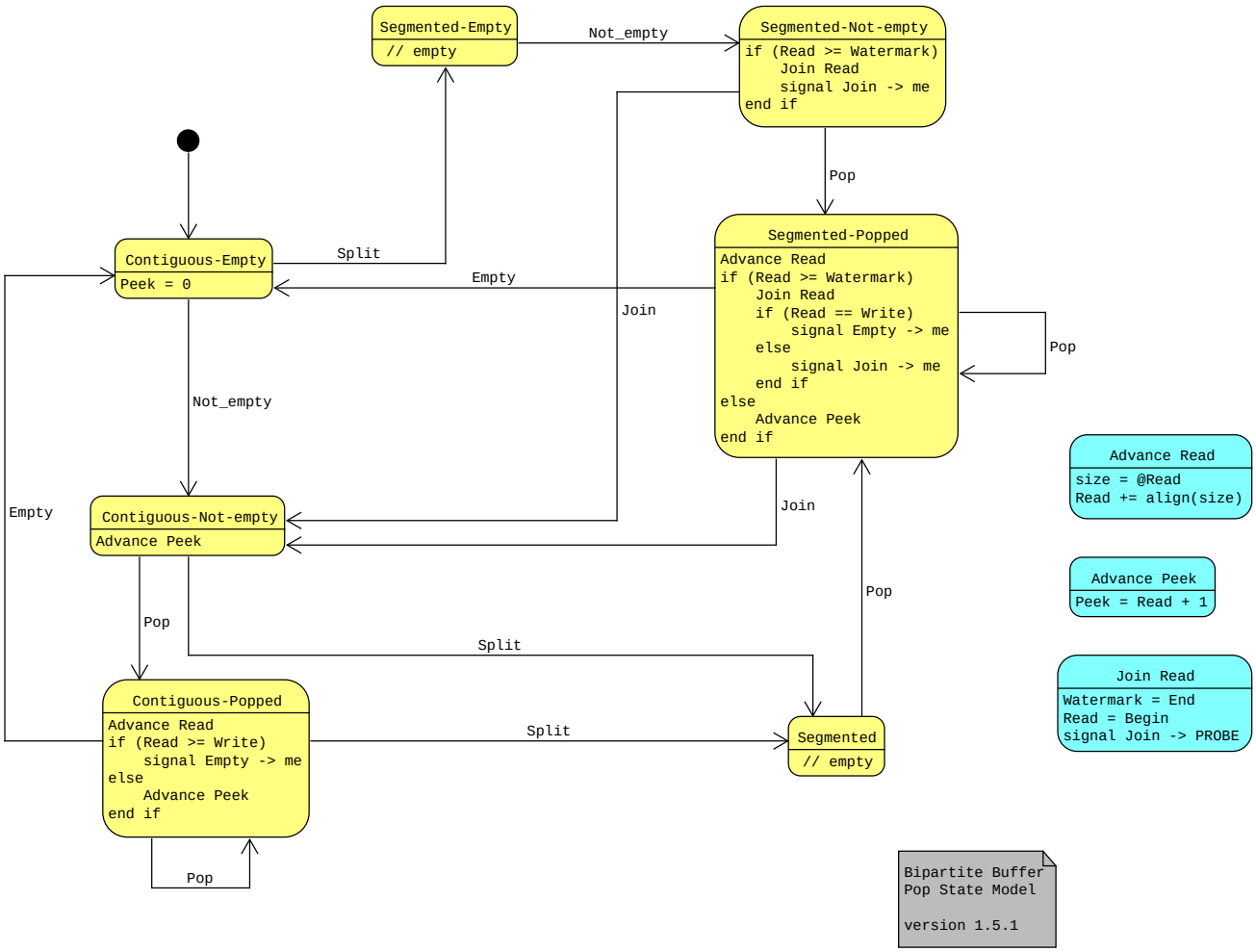

Figure 18.2: Bipartite Buffer Pop State Model

As discussed previously, the only operation on the read side of the buffer which affects the state of the buffer is to *pop* the buffer which discards the slot at the beginning of the queue. The *peek* operation is idempotent.

Much like the states of the probe state model, the pop state model is concerned about when the buffer is contiguous or segmented. The only allowed operation on the read side is to pop the buffer, so there are fewer states and the Pop event can cause transitions back to the same state. The pop state model determines when to join a segmented buffer back to be a contiguous one.

The state model starts in the **Contiguous-Empty** state. When the write side pushes data into the queue, the Not_empty event is signaled to inform the read side that there is something which can be popped. It may seem strange that a Split event is allowed when the buffer is empty, but recall the buffer is considered empty when Read is equal to Write and it is possible (after at least on cycle through the buffer) that the Read and Write pointers are near the end of the buffer so that the next allocation causes a split.

After the write side has pushed a slot, the machine transitions to the **Contiguous-Not-empty** state. The pop operation causes the transition to the **Contiguous-Popped** state. While the buffer is contiguous, pop operations advance the Read pointer to catch up with the Write pointer. Should they become equal, then the buffer has gone empty and this can happen without being split.

When the probe state model signals a Split, the transition paths switch to the segmented states. The primary difference here is that emptiness is determined by whether the Read pointer catches up with the Watermark pointer (as opposed to the Write pointer when the buffer was contiguous). When the Read pointer catches up with the Watermark pointer in a segmented buffer, the buffer is considered joined together again. Joining places the Watermark at the End of the buffer and wraps the Read pointer back to the Beginning.

The transition path from **Contiguous-Empty** to **Segmented-Empty** via Split, **Segmented-Empty** to **Segmented-Not-empty**

via Not_empty, and **Segmented-Not-empty** to **Contiguous-Not-empty** via Not_empty deserves additional commentary. The circumstances for this set of transitions correspond to an empty buffer where the Read and Write pointers are too close to the End of the buffer for the next allocation. So the probe state model splits the buffer. A buffer split only occurs as a result of a probe, so even through the buffer is split, it appears empty from the read side. When the write side pushes the slot, we must now consider the buffer as joined since there are no remaining allocated slots past the Read pointer. Joining then sets the Read pointer back to the Beginning which is where the most recently pushed slot is located.

Bip Buffer State Machine Context

We define a context so that the probe and pop state models can have their state machines interact.

```
<<bip_buffer: static data>>=
static
SSM_DEFINE_CONTEXT(bip_buffer_context) ;
```

Probe State Model Definition

The synchronous state machine implementation provides a number of macros and a process to use to define the necessary data structures to create a state model.

```
<<bip_buffer: include files>>=
#include "sync_sm.h"
```

The step required to define a state model for the probe side of the bip-buffer are:

Define a enumeration for the states.

```
<<bip_buffer: interface data type declarations>>=
typedef enum {
    prbs_Empty,
    prbs_Empty_Probed,
    prbs_Contiguous_Pushed,
    prbs_Contiguous_Probed,
    prbs_Contiguous_Full,
    prbs_Segmented,
    prbs_Segmented_Pushed,
    prbs_Segmented_Probed,
    prbs_Segmented_Full,
    prbs_Join_Pushed,
    prbs_Join_Probed,

    prbs_PROBE_STATE_COUNT          // last
} Bip_ProbeStates ;
```

Define the names of the states. This is useful for debugging.

```
<<bip_buffer: static data>>=
static const
SSM_DEFINE_STATE_NAMES(bip_probe_model, prbs_PROBE_STATE_COUNT) = {
    SSM_NAME_STATE(prbs_Empty, Empty),
    SSM_NAME_STATE(prbs_Empty_Probed, Empty-Probed),
    SSM_NAME_STATE(prbs_Contiguous_Pushed, Contiguous-Pushed),
    SSM_NAME_STATE(prbs_Contiguous_Probed, Contiguous-Probed),
    SSM_NAME_STATE(prbs_Contiguous_Full, Contiguous-Full),
    SSM_NAME_STATE(prbs_Segmented, Segmented),
    SSM_NAME_STATE(prbs_Segmented_Pushed, Segmented-Pushed),
    SSM_NAME_STATE(prbs_Segmented_Probed, Segmented-Probed),
```

```
    SSM_NAME_STATE(prbs_Segmented_Full, Segmented-Full),
    SSM_NAME_STATE(prbs_Join_Pushed, Join-Pushed),
    SSM_NAME_STATE(prbs_Join_Probed, Join-Probed),
} ;
```

Define a enumeration for the events.

```
<<bip_buffer: interface data type declarations>>=
typedef enum {
    prbe_Empty,
    prbe_Probe,
    prbe_Push,
    prbe_Full,
    prbe_Split,
    prbe_Join,

    prbe_PROBE_EVENT_COUNT          // last
} Bip_ProbeEvents ;
```

Define the names of the events in a similar manner as for the states.

```
<<bip_buffer: static data>>=
static const
SSM_DEFINE_EVENT_NAMES(bip_probe_model, prbe_PROBE_EVENT_COUNT) = {
    SSM_NAME_EVENT(prbe_Empty, Empty),
    SSM_NAME_EVENT(prbe_Probe, Probe),
    SSM_NAME_EVENT(prbe_Push, Push),
    SSM_NAME_EVENT(prbe_Full, Full),
    SSM_NAME_EVENT(prbe_Split, Split),
    SSM_NAME_EVENT(prbe_Join, Join),
} ;
```

Define the state activity table. This is an array of function pointers to the activities for each state. A `NULL` activity pointer means the activity is empty and no function call is made.

```
<<bip_buffer: static data>>=
static const
SSM_DEFINE_ACTIVITY_TABLE(bip_probe_model, prbs_PROBE_STATE_COUNT) = {
    SSM_DEFINE_ACTIVITY(prbs_Empty, NULL),
    SSM_DEFINE_ACTIVITY(prbs_Empty_Probed, prb_empty_probed_activity),
    SSM_DEFINE_ACTIVITY(prbs_Contiguous_Pushed, prb_contiguous_pushed_activity),
    SSM_DEFINE_ACTIVITY(prbs_Contiguous_Probed, prb_contiguous_probed_activity),
    SSM_DEFINE_ACTIVITY(prbs_Contiguous_Full, NULL),
    SSM_DEFINE_ACTIVITY(prbs_Segmented, NULL),
    SSM_DEFINE_ACTIVITY(prbs_Segmented_Pushed, prb_segmented_pushed_activity),
    SSM_DEFINE_ACTIVITY(prbs_Segmented_Probed, prb_segmented_probed_activity),
    SSM_DEFINE_ACTIVITY(prbs_Segmented_Full, NULL),
    SSM_DEFINE_ACTIVITY(prbs_Join_Pushed, NULL),
    SSM_DEFINE_ACTIVITY(prbs_Join_Probed, NULL),
} ;
```

Define the transition table. This is a `states` by `events` array. The elements of the array are the `new state` into which the machine transitions when a given event is received by a given state. An transition matrix element of `SSM_State_IG` means the transition is ignored. A element value of `SSM_State_CH` is a logical impossibility (“can’t happen”), is an error condition, and no transition takes place.

The following table shows the transition matrix for the probe state model.

Table 18.1: Probe State Model Transition Matrix

	Empty	Probe	Push	Full	Split	Join
Empty	CH	Empty-Probed	CH	CH	CH	CH
Empty-Probed	Empty	CH	Contiguous-Pushed	CH	CH	CH
Contiguous-Pushed	CH	Contiguous-Probed	CH	CH	CH	CH
Contiguous-Probed	CH	CH	Contiguous-Pushed	Contiguous-Full	Segmented	CH
Contiguous-Full	CH	Contiguous-Probed	CH	CH	CH	CH
Segmented	CH	CH	Segmented-Pushed	CH	CH	Join-Probed
Segmented-Pushed	CH	Segmented-Probed	CH	CH	CH	Join-Pushed
Segmented-Probed	CH	CH	Segmented-Pushed	Segmented-Full	CH	CH
Segmented-Full	CH	Segmented-Probed	CH	CH	CH	Join-Pushed
Join-Pushed	CH	Contiguous-Probed	CH	CH	CH	CH
Join-Probed	CH	CH	Contiguous-Pushed	CH	CH	CH

The following macro invocations define the transition array. *N.B.* there must be one invocation of `SSM_DEFINE_TRANSITION` for *every* cell in the transition matrix.

```
<<bip_buffer: static data>>=
static const
SSM_DEFINE_TRANS_TABLE(bip_probe_model, prbs_PROBE_STATE_COUNT, prbe_PROBE_EVENT_COUNT) = {
    SSM_DEFINE_TRANSITION(prbs_Empty, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Empty, prbe_Probe, prbs_Empty_Probed),
    SSM_DEFINE_TRANSITION(prbs_Empty, prbe_Push, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Empty, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Empty, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Empty, prbe_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(prbs_Empty_Probed, prbe_Empty, prbs_Empty),
    SSM_DEFINE_TRANSITION(prbs_Empty_Probed, prbe_Probe, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Empty_Probed, prbe_Push, prbs_Contiguous_Pushed),
    SSM_DEFINE_TRANSITION(prbs_Empty_Probed, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Empty_Probed, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Empty_Probed, prbe_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(prbs_Contiguous_Pushed, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Pushed, prbe_Probe, prbs_Contiguous_Probed),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Pushed, prbe_Push, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Pushed, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Pushed, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Pushed, prbe_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(prbs_Contiguous_Probed, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Probed, prbe_Probe, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Probed, prbe_Push, prbs_Contiguous_Pushed),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Probed, prbe_Full, prbs_Contiguous_Full),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Probed, prbe_Split, prbs_Segmented),
```

```
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Probed, prbe_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(prbs_Contiguous_Full, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Full, prbe_Probe, prbs_Contiguous_Probed),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Full, prbe_Push, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Full, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Full, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Contiguous_Full, prbe_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(prbs_Segmented, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented, prbe_Probe, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented, prbe_Push, prbs_Segmented_Pushed),
    SSM_DEFINE_TRANSITION(prbs_Segmented, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented, prbe_Join, prbs_Join_Probed),

    SSM_DEFINE_TRANSITION(prbs_Segmented_Pushed, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Pushed, prbe_Probe, prbs_Segmented_Probed),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Pushed, prbe_Push, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Pushed, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Pushed, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Pushed, prbe_Join, prbs_Join_Pushed),

    SSM_DEFINE_TRANSITION(prbs_Segmented_Probed, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Probed, prbe_Probe, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Probed, prbe_Push, prbs_Segmented_Pushed),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Probed, prbe_Full, prbs_Segmented_Full),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Probed, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Probed, prbe_Join, prbs_Join_Probed),

    SSM_DEFINE_TRANSITION(prbs_Segmented_Full, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Full, prbe_Probe, prbs_Segmented_Probed),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Full, prbe_Push, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Full, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Full, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Segmented_Full, prbe_Join, prbs_Join_Pushed),

    SSM_DEFINE_TRANSITION(prbs_Join_Pushed, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Join_Pushed, prbe_Probe, prbs_Contiguous_Probed),
    SSM_DEFINE_TRANSITION(prbs_Join_Pushed, prbe_Push, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Join_Pushed, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Join_Pushed, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Join_Pushed, prbe_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(prbs_Join_Probed, prbe_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Join_Probed, prbe_Probe, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Join_Probed, prbe_Push, prbs_Contiguous_Pushed),
    SSM_DEFINE_TRANSITION(prbs_Join_Probed, prbe_Full, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Join_Probed, prbe_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(prbs_Join_Probed, prbe_Join, SSM_State_CH),
} ;
```

Finally, the entire state model is constructed. The previous component definitions assembled into the required data structure.

```
<<bip_buffer: external data declarations>>=
extern const SSM_StateModel bip_probe_model ;
```

```
<<bip_buffer: external data definitions>>=
const
SSM_DEFINE_STATE_MODEL(bip_probe_model, prbs_Empty,
        prbs_PROBE_STATE_COUNT, prbe_PROBE_EVENT_COUNT) ;
```

Probe State Model Activities Implementation

The following section show the implementation of the state activities for the probe state model. We have placed the pseudo-code from each activity immediately before the "C" code for its implementation. Each state activity is placed in a separate function.

Empty-Probed Activity

Empty-Probed activity

```
alloc = align_probe(size)
avail = Watermark - Write
if (avail > alloc)
    Advance Probe(size)
else
    signal Empty -> me    // overflow
end if
```

Empty-Probed implementation

```
<<bip_buffer: static functions>>=
static void
prb_empty_probed_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, probe_sm) ;  // (1)
    size_t alloc = align_probe(bipbuf->probe_size) ;

    uintptr_t wr_pos = (uintptr_t)bipbuf->write ;
    uintptr_t wm_pos = (uintptr_t)bipbuf->watermark ;
    size_t avail = wm_pos - wr_pos ;

    if (avail > alloc) {
        advance_probe(bipbuf) ;
    } else {
        SSM_signalSelf(machine, prbe_Empty) ;
    }
}
```

(1) The argument to the state activity is the pointer to the state machine for the bip-buffer. To obtain the pointer to the bip-buffer itself, we use the old `CONTAINER_OF` macro to perform the underhanded address arithmetic which "up casts" from the state machine pointer to the containing bip-buffer data structure. This technique is used repeatedly in the state activities.

We need a header file to obtain a definition of the `CONTAINER_OF` macro.

```
<<bip_buffer: include files>>=
#include "useful.h"
```

Contiguous-Pushed Activity

Contiguous-Pushed activity

```
Advance Write
signal Not_empty -> POP
```

Contiguous-Pushed implementation

```
<<bip_buffer: static functions>>=
static void
prb_contiguous_pushed_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, probe_sm) ;

    advance_write(bipbuf) ;
    SSM_signal(&bipbuf->pop_sm, pope_Not_empty) ;
}
```

Contiguous-Probed Activity

Contiguous-Probed activity

```
alloc = align_probe(size)
avail = Watermark - Write
if (avail > alloc)
    Advance Probe(size)
else
    avail = Read - Begin
    if (avail > alloc)
        Watermark = Write
        Write = Begin
        Advance Probe(size)
        signal Split -> POP
        signal Split -> me
    else
        Probe = 0
        signal Full -> me
    end if
end if
```

Contiguous-Probed implementation

```
<<bip_buffer: static functions>>=
static void
prb_contiguous_probed_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, probe_sm) ;
    size_t alloc = align_probe(bipbuf->probe_size) ;

    uintptr_t wr_pos = (uintptr_t)bipbuf->write ;
    uintptr_t wm_pos = (uintptr_t)bipbuf->watermark ;
    size_t avail = wm_pos - wr_pos ;

    if (avail >= alloc) {
        advance_probe(bipbuf) ;
    } else {
        uintptr_t rd_pos = (uintptr_t)bipbuf->read ;
        uintptr_t begin_pos = (uintptr_t)bipbuf->begin ;
        avail = rd_pos - begin_pos ;
        if (avail > alloc) {
            bipbuf->watermark = bipbuf->write ;
            bipbuf->write = bipbuf->begin ;
            advance_probe(bipbuf) ;
            SSM_signal(&bipbuf->pop_sm, pope_Split) ;
            SSM_signalSelf(machine, prbe_Split) ;
        } else {
```

```
            bipbuf->probe = NULL ;
            SSM_signalSelf(machine, prbe_Full) ;
        }
    }
}
```

Segmented-Pushed Activity

Segmented-Pushed activity

```
Advance Write
signal Not_empty -> POP
```

Segmented-Pushed implementation

```
<<bip_buffer: static functions>>=
static void
prb_segmented_pushed_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, probe_sm) ;

    advance_write(bipbuf) ;
    SSM_signal(&bipbuf->pop_sm, pope_Not_empty) ;
}
```

Segmented-Probed Activity

Segmented-Probed activity

```
alloc = align(size + 1)
avail = Read - Write
if (avail > alloc)
    Advance Probe(size)
else
    Probe = 0
    signal Full -> me
end if
```

Segmented-Probed implementation

```
<<bip_buffer: static functions>>=
static void
prb_segmented_probed_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, probe_sm) ;
    size_t alloc = align_probe(bipbuf->probe_size) ;

    uintptr_t rd_pos = (uintptr_t)bipbuf->read ;
    uintptr_t wr_pos = (uintptr_t)bipbuf->write ;
    size_t avail = rd_pos - wr_pos ;

    if (avail > alloc) {
        advance_probe(bipbuf) ;
    } else {
        bipbuf->probe = NULL ;
        SSM_signalSelf(machine, prbe_Full) ;
    }
}
```

Align Probe Method

Align Probe implementation

```
<<bip_buffer: static inline functions>>=
static inline size_t
align_probe(
    size_t probe_size)
{
    unsigned const align_mask = sizeof(size_t) - 1 ;
    return (probe_size + sizeof(size_t) + align_mask) & ~align_mask ;
}
```

Advance Probe Method

Advance Probe method

```
@Write = size
Probe = Write + 1
```

Advance Probe implementation

```
<<bip_buffer: static inline functions>>=
static inline void
advance_probe(
    BipBuffer *const bipbuf)
{
    *(size_t *)bipbuf->write = bipbuf->probe_size ;
    bipbuf->probe = (void *)((uintptr_t)bipbuf->write + sizeof(size_t)) ;
}
```

Advance Write Method

Advance Write method

```
size = @Write
Write += align(size)
```

Advance Write method

```
<<bip_buffer: static inline functions>>=
static inline void
advance_write(
    BipBuffer *const bipbuf)
{
    size_t probe_size = *(size_t *)bipbuf->write ;
    bipbuf->write = (void *)((uintptr_t)bipbuf->write + align_probe(probe_size)) ;
}
```

Pop State Model Implementation

Following the same procedure as for the Probe state model, we define the Pop state model.

```
<<bip_buffer: interface data type declarations>>=
typedef enum {
    pops_Contiguous_Empty,
    pops_Contiguous_Not_empty,
    pops_Contiguous_Popped,
    pops_Segmented,
    pops_Segmented_Empty,
    pops_Segmented_Not_empty,
    pops_Segmented_Popped,

    pops_POP_STATE_COUNT          // last
} Bip_PopStates ;
```

```
<<bip_buffer: static data>>=
static const
SSM_DEFINE_STATE_NAMES(bip_pop_model, pops_POP_STATE_COUNT) = {
    SSM_NAME_STATE(pops_Contiguous_Empty, Empty),
    SSM_NAME_STATE(pops_Contiguous_Not_empty, Contiguous-Not-empty),
    SSM_NAME_STATE(pops_Contiguous_Popped, Contiguous-Popped),
    SSM_NAME_STATE(pops_Segmented, Segmented),
    SSM_NAME_STATE(pops_Segmented_Empty, Segmented-Empty),
    SSM_NAME_STATE(pops_Segmented_Not_empty, Segmented-Not-empty),
    SSM_NAME_STATE(pops_Segmented_Popped, Segmented-Popped),
} ;
```

```
<<bip_buffer: interface data type declarations>>=
typedef enum {
    pope_Empty,
    pope_Not_empty,
    pope_Pop,
    pope_Split,
    pope_Join,

    pope_POP_EVENT_COUNT          // last
} Bip_PopEvents ;
```

```
<<bip_buffer: static data>>=
static const
SSM_DEFINE_EVENT_NAMES(bip_pop_model, pope_POP_EVENT_COUNT) = {
    SSM_NAME_EVENT(pope_Empty, Empty),
    SSM_NAME_EVENT(pope_Not_empty, Not_empty),
    SSM_NAME_EVENT(pope_Pop, Pop),
    SSM_NAME_EVENT(pope_Split, Split),
    SSM_NAME_EVENT(pope_Join, Join),
} ;
```

```
<<bip_buffer: static data>>=
static const
SSM_DEFINE_ACTIVITY_TABLE(bip_pop_model, pops_POP_STATE_COUNT) = {
    SSM_DEFINE_ACTIVITY(pops_Contiguous_Empty, pop_contiguous_empty_activity),
    SSM_DEFINE_ACTIVITY(pops_Contiguous_Not_empty, pop_contiguous_not_empty_activity),
    SSM_DEFINE_ACTIVITY(pops_Contiguous_Popped, pop_contiguous_popped_activity),
    SSM_DEFINE_ACTIVITY(pops_Segmented, NULL),
    SSM_DEFINE_ACTIVITY(pops_Segmented_Empty, NULL),
    SSM_DEFINE_ACTIVITY(pops_Segmented_Not_empty, pop_segmented_not_empty_activity),
    SSM_DEFINE_ACTIVITY(pops_Segmented_Popped, pop_segmented_popped_activity),
} ;
```

The following table shows the transition matrix for the pop state model.

Table 18.2: Pop State Model Transition Matrix

	Empty	Not_empty	Pop	Split	Join
Contiguous-Empty	CH	Contiguous-Not-empty	CH	Segmented-Empty	CH
Contiguous-Not-empty	CH	IG	Contiguous-Popped	Segmented	CH
Contiguous-Popped	Contiguous-Empty	IG	Contiguous-Popped	Segmented	CH
Segmented	CH	IG	Segmented-Popped	CH	CH
Segmented-Empty	CH	Segmented-Not-empty	CH	CH	CH
Segmented-Not-empty	CH	IG	Segmented-Popped	CH	CH
Segmented-Popped	Contiguous-Empty	IG	Segmented-Popped	CH	Contiguous-Not-empty

```
<<bip_buffer: static data>>=
static const
SSM_DEFINE_TRANS_TABLE(bip_pop_model, pops_POP_STATE_COUNT, pope_POP_EVENT_COUNT) = {
    SSM_DEFINE_TRANSITION(pops_Contiguous_Empty, pope_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Empty, pope_Not_empty, pops_Contiguous_Not_empty) ↩
        ,
    SSM_DEFINE_TRANSITION(pops_Contiguous_Empty, pope_Pop, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Empty, pope_Split, pops_Segmented_Empty),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Empty, pope_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(pops_Contiguous_Not_empty, pope_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Not_empty, pope_Not_empty, SSM_State_IG),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Not_empty, pope_Pop, pops_Contiguous_Popped),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Not_empty, pope_Split, pops_Segmented),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Not_empty, pope_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(pops_Contiguous_Popped, pope_Empty, pops_Contiguous_Empty),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Popped, pope_Not_empty, SSM_State_IG),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Popped, pope_Pop, pops_Contiguous_Popped),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Popped, pope_Split, pops_Segmented),
    SSM_DEFINE_TRANSITION(pops_Contiguous_Popped, pope_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(pops_Segmented, pope_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Segmented, pope_Not_empty, SSM_State_IG),
    SSM_DEFINE_TRANSITION(pops_Segmented, pope_Pop, pops_Segmented_Popped),
    SSM_DEFINE_TRANSITION(pops_Segmented, pope_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Segmented, pope_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(pops_Segmented_Empty, pope_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Segmented_Empty, pope_Not_empty, pops_Segmented_Not_empty),
    SSM_DEFINE_TRANSITION(pops_Segmented_Empty, pope_Pop, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Segmented_Empty, pope_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Segmented_Empty, pope_Join, SSM_State_CH),

    SSM_DEFINE_TRANSITION(pops_Segmented_Not_empty, pope_Empty, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Segmented_Not_empty, pope_Not_empty, SSM_State_IG),
    SSM_DEFINE_TRANSITION(pops_Segmented_Not_empty, pope_Pop, pops_Segmented_Popped),
    SSM_DEFINE_TRANSITION(pops_Segmented_Not_empty, pope_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Segmented_Not_empty, pope_Join, pops_Contiguous_Not_empty),
```

```
    SSM_DEFINE_TRANSITION(pops_Segmented_Popped, pope_Empty, pops_Contiguous_Empty),
    SSM_DEFINE_TRANSITION(pops_Segmented_Popped, pope_Not_empty, SSM_State_IG),
    SSM_DEFINE_TRANSITION(pops_Segmented_Popped, pope_Pop, pops_Segmented_Popped),
    SSM_DEFINE_TRANSITION(pops_Segmented_Popped, pope_Split, SSM_State_CH),
    SSM_DEFINE_TRANSITION(pops_Segmented_Popped, pope_Join, pops_Contiguous_Not_empty),
} ;
```

```
<<bip_buffer: external data declarations>>=
extern const SSM_StateModel bip_pop_model ;
```

```
<<bip_buffer: external data definitions>>=
const
SSM_DEFINE_STATE_MODEL(bip_pop_model, pops_Contiguous_Empty,
        pops_POP_STATE_COUNT, pope_POP_EVENT_COUNT) ;
```

Pop State Model Activities Implementation

Contiguous-Empty Activity

Contiguous-Empty activity

```
Peek = 0
```

Contiguous-Empty implementation

```
<<bip_buffer: static functions>>=
static void
pop_contiguous_empty_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, pop_sm) ;

    bipbuf->peek = NULL ;
}
```

Contiguous-Not-Empty Activity

Contiguous-Not-empty activity

```
Advance Peek
```

Contiguous-Not-empty implementation

```
<<bip_buffer: static functions>>=
static void
pop_contiguous_not_empty_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, pop_sm) ;

    advance_peek(bipbuf) ;
}
```

Contiguous-Popped Activity

Contiguous-Popped activity

```
Advance Read
if (Read >= Write)
    signal Empty -> me
else
    Advance Peek
end if
```

Contiguous-Popped implementation

```
<<bip_buffer: static functions>>=
static void
pop_contiguous_popped_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, pop_sm) ;

    advance_read(bipbuf) ;
    if (bipbuf->read >= bipbuf->write) {
        SSM_signalSelf(machine, pope_Empty) ;
    } else {
        advance_peek(bipbuf) ;
    }
}
```

Segmented-Not-empty Activity

Segmented-Not-empty activity

```
Join Read
signal Not_empty -> me
```

Segmented-Not-empty implementation

```
<<bip_buffer: static functions>>=
static void
pop_segmented_not_empty_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, pop_sm) ;

    if (bipbuf->read >= bipbuf->watermark) {
        join_read(bipbuf) ;
        SSM_signalSelf(machine, pope_Join) ;
    }
}
```

Segmented-Popped Activity

Segmented-Popped activity

```
Advance Read
if (Read >= Watermark)
    Join Read
    if (Read == Write)
        signal Empty -> me
```

```
    else
        signal Not_empty -> me
    end if
else
    Advance Peek
end if
```

Segmented-Popped implementation

```
<<bip_buffer: static functions>>=
static void
pop_segmented_popped_activity(
    SSM_Machine *const machine)
{
    BipBuffer *const bipbuf = CONTAINER_OF(machine, BipBuffer, pop_sm) ;

    advance_read(bipbuf) ;
    if (bipbuf->read >= bipbuf->watermark) {
        join_read(bipbuf) ;
        SSM_signalSelf(machine,
                bipbuf->read == bipbuf->write ? pope_Empty : pope_Join) ;
    } else {
        advance_peek(bipbuf) ;
    }
}
```

Advance Read Method

Advance Read method

```
size = @Read
Read += align(size)
```

Advance Read method

```
<<bip_buffer: static inline functions>>=
static inline void
advance_read(
    BipBuffer *const bipbuf)
{
    size_t probe_size = *(size_t *)bipbuf->read ;
    bipbuf->read = (void *)((uintptr_t)bipbuf->read + align_probe(probe_size)) ;
}
```

Advance Peek Method

Advance Peek method

```
Peek = Read + 1
```

Advance Peek method

```
<<bip_buffer: static inline functions>>=
static inline void
advance_peek(
    BipBuffer *const bipbuf)
{
    bipbuf->peek = (void *)((uintptr_t)bipbuf->read + sizeof(size_t)) ;
}
```

Join Read Method

Join Read method

```
Watermark = End
Read = Begin
signal Join -> write_sm
```

Join Read method

```
<<bip_buffer: static inline functions>>=
static inline void
join_read(
    BipBuffer *const bipbuf)
{
    bipbuf->watermark = bipbuf->end ;
    bipbuf->read = bipbuf->begin ;
    SSM_signal(&bipbuf->probe_sm, prbe_Join) ;
}
```

Bipartite Buffer Code Layout

Bipartite buffer header file

```
<<bip_buffer.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Bipartite Buffer Interface
 *--
 */
#ifndef BIPBUFFER_H_
#define BIPBUFFER_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdalign.h>
#include <inttypes.h>
<<bip_buffer: include files>>
/*
 * Constants
 */
<<bip_buffer: constants>>
/*
 * Macros
 */
<<bip_buffer: macros>>
/*
 * Data Type Declarations
 */
<<bip_buffer: interface data type declarations>>
/*
 * External Data Declarations
```

```
 */
<<bip_buffer: external data declarations>>
/*
 * External Functions
 */
<<bip_buffer: external function declarations>>

#endif /* BIPBUFFER_H_ */
```

Bipartite buffer code file

```
<<bip_buffer.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Bipartite Buffer Functions
 *--
 */

/*
 * Include files
 */
#include <inttypes.h>
#include <assert.h>
#include "rtcheck.h"
#include "bip_buffer.h"
/*
 * Static Inline Functions
 */
<<bip_buffer: static inline functions>>
/*
 * Static Functions
 */
<<bip_buffer: static functions>>
/*
 * Static Data
 */
<<bip_buffer: static data>>
/*
 * External Data Definitions
 */
<<bip_buffer: external data definitions>>
/*
 * External Functions
 */
<<bip_buffer: external function definitions>>
```

Testing

To test the bip buffer functions, we use the Unity testing framework. The Unity framework is small, simple, pure “C”, and can be used in low resource contexts.

Defining a Buffer

```
<<bip_buffer test: static data>>=
BIP_DEFINE_BUFFER(test_bipbuf, 128) ;
```

Unity test cases

To use the Unity framework, we need the header file.

```
<<bip_buffer test: include files>>=
#include <stdio.h>
#ifdef USE_SEGGER_RTT
#   include "SEGGER_RTT.h"
#endif /* USE_SEGGER_RTT */
#include "unity.h"
```

```
<<bip_buffer test: static functions>>=
void
setUp(void)
{
}
void
tearDown(void)
{
}
```

```
<<bip_buffer test: static functions>>=
void
test_probe_1(void) {
    void *frag = bip_probe(&test_bipbuf, 96) ;
    TEST_ASSERT_NOT_NULL(frag) ;
    bip_push(&test_bipbuf) ;
}
```

```
<<bip_buffer test: static functions>>=
void
test_probe_2(void) {
    void *frag = bip_probe(&test_bipbuf, 16) ;
    TEST_ASSERT_NOT_NULL(frag) ;
    bip_push(&test_bipbuf) ;
}
```

```
<<bip_buffer test: static functions>>=
void
test_probe_3(void) {
    void *frag = bip_probe(&test_bipbuf, 100) ;
    TEST_ASSERT_NULL(frag) ;
}
```

```
<<bip_buffer test: static functions>>=
void
test_peek_1(void) {
    size_t len ;
    void *frag = bip_peek(&test_bipbuf, &len) ;
    TEST_ASSERT_NOT_NULL(frag) ;
    TEST_ASSERT_EQUAL_UINT32(96, len) ;
    void *next_frag = bip_pop(&test_bipbuf, &len) ;
    TEST_ASSERT_NOT_NULL(next_frag) ;
    TEST_ASSERT_EQUAL_UINT32(16, len) ;
}
```

```
<<bip_buffer test: static functions>>=
void
test_peek_2(void) {
    size_t len ;
    void *frag = bip_peek(&test_bipbuf, &len) ;
    TEST_ASSERT_NOT_NULL(frag) ;
    TEST_ASSERT_EQUAL_UINT32(16, len) ;
    void *next_frag = bip_pop(&test_bipbuf, &len) ;
    TEST_ASSERT_NULL(next_frag) ;
}
```

```
<<bip_buffer test: static functions>>=
void
test_probe_4(void) {
    void *frag = bip_probe(&test_bipbuf, 50) ;
    TEST_ASSERT_NOT_NULL(frag) ;
    bip_push(&test_bipbuf) ;
}
```

```
<<bip_buffer test: static functions>>=
void
test_peek_3(void) {
    size_t len ;
    void *frag = bip_peek(&test_bipbuf, &len) ;
    TEST_ASSERT_NOT_NULL(frag) ;
    TEST_ASSERT_EQUAL_UINT32(50, len) ;
    void *next_frag = bip_pop(&test_bipbuf, &len) ;
    TEST_ASSERT_NULL(next_frag) ;
}
```

```
<<bip_buffer test: static functions>>=
void
test_probe_5(void) {
    size_t frag_size = 4 ;
    void *frag ;

    for (frag = bip_probe(&test_bipbuf, frag_size) ; frag != NULL ;
            frag = bip_probe(&test_bipbuf, frag_size)) {
        TEST_ASSERT_NOT_NULL(frag) ;
        bip_push(&test_bipbuf) ;

        frag_size *= 2 ;
    }

    frag_size = 4 ;
    size_t actual ;
    for (frag = bip_peek(&test_bipbuf, &actual) ; frag != NULL ;
            frag = bip_pop(&test_bipbuf, &actual)) {
        TEST_ASSERT_NOT_NULL(frag) ;
        TEST_ASSERT_EQUAL_UINT32(actual, frag_size) ;

        frag_size *= 2 ;
    }
}
```

```
<<bip_buffer test: static functions>>=
void
test_probe_6(void) {
    static unsigned const pushes[] = {3, 1, 4, 6} ;
    static unsigned const pops[] = {2, 1, 4, 7} ;
```

```
    static size_t const sizes[] = {21, 31, 8, 5} ;

    unsigned const *push = pushes ;
    unsigned const *const push_end = pushes + (sizeof(pushes) / sizeof(pushes[0])) ;
    unsigned const *pop = pops ;
    size_t const *size = sizes ;

    bip_reset(&test_bipbuf) ;

    while (push < push_end) {
        size_t frag_size = *size++ ;
        for (unsigned push_count = *push++ ; push_count != 0 ; push_count--) {
            void *frag = bip_probe(&test_bipbuf, frag_size) ;
            TEST_ASSERT_NOT_NULL(frag) ;
            bip_push(&test_bipbuf) ;
        }

        for (unsigned pop_count = *pop++ ; pop_count != 0 ; pop_count--) {
            size_t actual ;
            void *frag = bip_peek(&test_bipbuf, &actual) ;
            printf("peeked %u bytes\n", actual) ;
            TEST_ASSERT_NOT_NULL(frag) ;
            bip_pop(&test_bipbuf, NULL) ;
        }
    }
}
```

```
<<bip_buffer test: main function>>=
int
main(
    int argc,
    char **argv)
{
#       ifdef USE_SEGGER_RTT
    SEGGER_RTT_Init() ;
#       endif /* USE_SEGGER_RTT */

    printf("beginning test run\n") ;

#       ifdef DEBUG_BUILD
    SSM_logEnable(&test_bipbuf.probe_sm, true) ;
    SSM_logEnable(&test_bipbuf.pop_sm, true) ;
#       endif /* DEBUG_BUILD */

    UNITY_BEGIN() ;

    RUN_TEST(test_probe_1) ;
    RUN_TEST(test_probe_2) ;
    RUN_TEST(test_probe_3) ;
    RUN_TEST(test_peek_1) ;
    RUN_TEST(test_peek_2) ;
    RUN_TEST(test_probe_4) ;
    RUN_TEST(test_peek_3) ;
    RUN_TEST(test_probe_5) ;
    RUN_TEST(test_probe_6) ;

    return UNITY_END() ;
}
```

Bipartite buffer testing file

```
<<bip-buffer-test.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   Bipartite Buffer Unit Tests
 *--
 */

/*
 * Include files
 */
<<bip_buffer test: include files>>
#include <inttypes.h>
#include "bip_buffer.h"
/*
 * Static Data
 */
<<bip_buffer test: static data>>
/*
 * Static Functions
 */
<<bip_buffer test: static functions>>
/*
 * External Functions
 */
<<bip_buffer test: main function>>
```

Chapter 19

Block Allocator

We have chosen *not* to use a system-wide heap for dynamic memory allocation. We still have need for dynamic memory allocation, but satisfy that requirement by using distinct memory pools for each data type. The code in this section is a supports block allocation out of an array whose size is known at compile time.

Allocator Meta-data

```
<<block alloc: data type declarations>>=
typedef struct {
    void *storageStart ;
    void *storageFinish ;
    void *storageLast ;
    uint32_t *allocStatus ;
    unsigned allocCount ;
    size_t blockSize ;
} BKAL_ControlBlock ;
```

```
<<block alloc: macros>>=
#define DEFINE_BLOCK_ALLOCATOR(name, type, count)                          \
    static_assert(count > 0, "allocation pool count must be positive") ;   \
    static type name ## __POOL[count] ;                                    \
    static uint32_t name ## __status[(count + 31) / 32] ;                  \
    static BKAL_ControlBlock name = {                                      \
        .storageStart = name ## __POOL,                                    \
        .storageFinish = name ## __POOL + count,                           \
        .storageLast = name ## __POOL + count - 1,                         \
        .allocStatus = name ## __status,                                   \
        .allocCount = 0,                                                   \
        .blockSize = sizeof(type),                                         \
    }
```

Allocating a Block

```
<<block alloc: external function declarations>>=
void *
blk_allocate(
    BKAL_ControlBlock *acb) ;
```

ab
A pointer to an allocator control block from which a block is to be allocated.

`blk_allocate` searches for an unused memory block in the memory pool described by `ab`. It returns a pointer to the allocated memory if successful and `NULL` if no memory is available in the class pool.

The allocation algorithm is a simple sequential search starting at the last location that was allocated.

```
<<block alloc: external function definitions>>=
void *
blk_allocate(
    BKAL_ControlBlock *acb)
{
    assert(acb != NULL) ;
    assert(acb->storageLast < acb->storageFinish) ;
    /*
     * Search for an empty slot in the pool. Start at the next location after
     * where we last allocated a block.
     */
    void *block = acb->storageLast ;
    do {
        block = blkNextBlock(acb, block) ;
        if (!isBlkAllocated(acb, block)) {
            setBlkAllocated(acb, block) ;
            acb->storageLast = block ;
            acb->allocCount += 1 ;
            return memset(block, 0, acb->blockSize) ;
        }
    } while (block != acb->storageLast) ;                    // ❶

    return NULL ;
}
```

❶ If we wrap all the way around to where we started and still did not find an unallocated block, then we have run out of space! That condition is indicated by returning `NULL`.

Finding the next element in the instance storage array involves performing the pointer arithmetic modulo the size of the array. Since the pool is allocated in a contiguous block of memory, we must wrap around the iterator when it passes the end of the storage pool. That is accomplished with the `blkNextBlock()` function.

```
<<block alloc: static inline functions>>=
static inline void *
blkNextBlock(
    BKAL_ControlBlock *acb,
    void *block)
{
    block = (void *)((uintptr_t)block + acb->blockSize) ; // ❶
    if (block >= acb->storageFinish) { // ❷
        block = acb->storageStart ;
    }
    return block ;
}
```

❶ Since the size of instance varies from class to class, we must take over scaling the pointer arithmetic by the size of the instance.

❷ Perform the wrap around if we cross over the boundary of the storage array.

```
<<block alloc: static inline functions>>=
static inline size_t
blkToIndex(
    BKAL_ControlBlock *acb,
    void *block)
{
    assert(acb != NULL) ;
    assert(block != NULL) ;
    assert(block >= acb->storageStart && block < acb->storageFinish) ;

    return ((uintptr_t)block - (uintptr_t)acb->storageStart) / acb->blockSize ;
}
```

```
<<block alloc: static inline functions>>=
static inline bool
isBlkAllocated(
    BKAL_ControlBlock *acb,
    void *block)
{
    size_t block_index = blkToIndex(acb, block) ;
    uint32_t status_bit = block_index % 32 ;
    uint32_t status_mask = btwd_bit_mask(status_bit) ;
    uint32_t status_word = acb->allocStatus[block_index / 32] ;

    return (status_word & status_mask) != 0 ;
}
```

```
<<block alloc: static inline functions>>=
static inline void
setBlkAllocated(
    BKAL_ControlBlock *acb,
    void *block)
{
    size_t block_index = blkToIndex(acb, block) ;
    uint32_t status_bit = block_index % 32 ;
    uint32_t status_mask = btwd_bit_mask(status_bit) ;
    uint32_t *status_block_ref = &acb->allocStatus[block_index / 32] ;  // ❶
    *status_block_ref |= status_mask ;
}
```

❶ Note the use of a reference to the allocation status word. We want to modify the value in place and have no use for allocation word value itself.

```
<<block alloc: static inline functions>>=
static inline void
clearBlkAllocated(
    BKAL_ControlBlock *acb,
    void *block)
{
    size_t block_index = blkToIndex(acb, block) ;
    uint32_t status_bit = block_index % 32 ;
    uint32_t status_mask = btwd_bit_mask(status_bit) ;
    uint32_t *status_block_ref = &acb->allocStatus[block_index / 32] ;  // ❶
    *status_block_ref &= ~status_mask ;
}
```

```
<<block alloc: external function declarations>>=
void
blk_free(
    BKAL_ControlBlock *acb,
    void *block) ;
```

```
<<block alloc: external function definitions>>=
void
blk_free(
    BKAL_ControlBlock *acb,
    void *block)
{
    assert(acb != NULL) ;
    assert(block != NULL) ;
    assert(block >= acb->storageStart && block < acb->storageFinish) ;
    assert(isBlkAllocated(acb, block)) ;

    if (block == NULL ||
            block < acb->storageStart ||
            block >= acb->storageFinish ||
            !isBlkAllocated(acb, block)) {
        sys_panic("%s: illegal attempt to free block\n", __func__) ;
    }

    clearBlkAllocated(acb, block) ;
}
```

```
<<block alloc: external function declarations>>=
size_t
blk_ref_to_index(
    BKAL_ControlBlock *acb,
    void *block) ;
```

```
<<block alloc: external function definitions>>=
size_t
blk_ref_to_index(
    BKAL_ControlBlock *acb,
    void *block)
{
    return blkToIndex(acb, block) ;
}
```

```
<<block alloc: external function declarations>>=
void *
blk_index_to_ref(
    BKAL_ControlBlock *acb,
    size_t blk_id) ;
```

```
<<block alloc: external function definitions>>=
void *
blk_index_to_ref(
    BKAL_ControlBlock *acb,
    size_t blk_id)
{
    assert(acb != NULL) ;

    void *block = (void *)((uintptr_t)acb->storageStart + acb->blockSize * blk_id) ;
    assert(block >= acb->storageStart && block < acb->storageFinish) ;
```

```
    return isBlkAllocated(acb, block) ? block : NULL ;
}
```

```
<<block alloc: data type declarations>>=
typedef struct {
    void *instance ;
    BKAL_ControlBlock *allocator ;
} BKAL_Iterator ;
```

```
<<block alloc: external function declarations>>=
void
blk_iterator_start(
    BKAL_Iterator *iter,
    BKAL_ControlBlock *acb) ;
```

```
<<block alloc: external function definitions>>=
void
blk_iterator_start(
    BKAL_Iterator *iter,
    BKAL_ControlBlock *acb)
{
    assert(iter != NULL) ;
    assert(acb != NULL) ;

    iter->allocator = acb ;
    iter->instance = NULL ;
    blk_iterator_next(iter) ;
}
```

```
<<block alloc: external function declarations>>=
bool
blk_iterator_more(
    BKAL_Iterator *iter) ;
```

```
<<block alloc: external function definitions>>=
bool
blk_iterator_more(
    BKAL_Iterator *iter)
{
    assert(iter != NULL) ;

    return iter->instance < iter->allocator->storageFinish ;
}
```

```
<<block alloc: external function declarations>>=
void *
blk_iterator_get(
    BKAL_Iterator *iter) ;
```

```
<<block alloc: external function definitions>>=
void *
blk_iterator_get(
    BKAL_Iterator *iter)
{
    assert(iter != NULL) ;

    return iter->instance ;
}
```

```
<<block alloc: external function declarations>>=
void
blk_iterator_next(
    BKAL_Iterator *iter) ;
```

```
<<block alloc: external function definitions>>=
void
blk_iterator_next(
    BKAL_Iterator *iter)
{
    assert(iter != NULL) ;

    for (iter->instance = iter->allocator->storageStart ;
            !isBlkAllocated(iter->allocator, iter->instance) &&
            iter->instance < iter->allocator->storageFinish ;
            iter->instance = blkNextBlock(iter->allocator, iter->instance)) {
    }
}
```

Code Layout

In literate programming terminology, a *chunk* is a named part of the final program. The program chunks form a tree and the root of that tree is given a name. We follow the convention of naming the root the same as the output file name. The process of extracting the program tree formed by the chunks is called *tangle*. By the default the program, **atangle**, extracts the root chunk to produce the "C" source file.

Header file

```
<<block_alloc.h>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   General purpose block allocator.
 *--
 */
#ifndef BLOCK_ALLOC_H_
#define BLOCK_ALLOC_H_

/*
 * Include files
 */
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
/*
 * Macros
 */
<<block alloc: macros>>
/*
 * Data Type Declarations
 */
```

```
<<block alloc: data type declarations>>
/*
 * External Declarations
 */
<<block alloc: external function declarations>>
#endif /* BLOCK_ALLOC_H_ */
```

Source file

```
<<block_alloc.c>>=
<<edit warning>>
<<copyright info>>
/*
 *++
 * Project:
 *   Bottom Up
 *
 * Module:
 *   General purpose block allocator.
 *--
 */

/*
 * Include files
 */
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "sys_svc_req.h"
#include "bit_twiddle.h"
#include "block_alloc.h"
/*
 * Static Inline Functions
 */
<<block alloc: static inline functions>>
/*
 * External Functions
 */
<<block alloc: external function definitions>>
```

Part V

Supplemental materials

Bibliography

Books

[1] [defguide] Joseph Yiu, The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors, Elsevier (2014), ISBN 13:978-0-12-408082-9.

[2] [mb-xuml] Stephen J. Mellor and Marc J. Balcer, Executable UML: a foundation for model-driven architecture, Addison-Wesley (2002), ISBN 0-201-74804-5.

[3] [rs-xuml] Chris Raistrick, Paul Francis, John Wright, Colin Carter and Ian Wilkie, Model Driven Architecture with Executable UML, Cambridge University Press (2004), ISBN 0-521-53771-1.

[4] [mtoc] Leon Starr, Andrew Mangogna and Stephen Mellor, Models to Code: With No Mysterious Gaps, Apress (2017), ISBN 978-1-4842-2216-4

[5] [ls-build], Leon Starr, How to Build Shlaer-Mellor Object Models, Yourdon Press (1996), ISBN 0-13-207663-2.

[6] [sm-data] Sally Shlaer and Stephen J. Mellor, Object Oriented Systems Analysis: Modeling the World in Data, Prentice-Hall (1988), ISBN 0-13-629023-X.

[7] [sm-states] Sally Shlaer and Stephen J. Mellor, Object Lifecycles: Modeling the World in States, Prentice-Hall (1992), ISBN 0-13-629940-7.

[8] [halbwachs] Halbwachs, Nicolas, Synchronous programming of reactive systems, Springer Science+Business Media (1993), ISBN 978-1-4419-5133-5.

[9] [brooks] Brooks, Frederick, P. Jr., The Mythical Man-Month, Addison-Wesley (1995), ISBN 0-201-83595-9.

Articles

[10] [ls-articulate] Leon Starr, How to Build Articulate UML Class Models, 2008, http://www.modelint.com/how-to-build-articulate-uml-class-models/

[11] [ls-time] Leon Starr, Time and Synchronization in Executable UML, 2008, http://www.modelint.com/time-and-synchronization-in-executable-uml/

[12] [mosely-marks] Ben Mosely and Peter Marks, Out of the Tar Pit, 2006, http://curtclifton.net/papers/-MoseleyMarks06a.pdf

[13] [foote-yoder] Brian Foote and Joseph Yoder, Big Ball of Mud, 1999, http://www.laputan.org/mud/mud.html

Appendix A

List of Terms

AAPCS
ARM® Architecture Procedure Call Standard

ABI
application binary interface

ADC
analog to digital converter

BCD
binary coded decimal

BLE
Bluetooth low energy

BSS
block started by symbol — an historical assembler pseudo-operation carried forward into current usage to mean an area in memory where space for a variable has been allocated but no initial value has been given

CE
current era of calendar time

DAC
digital to analog converter

DMA
direct memory access

DSP
digital signal processing

DWT
data watchpoint and trace unit of the ARM Cortex processor

FPU
floating point unit

GPIO
general purpose input / output

KiB
kibibyte — equals 1024_{10} bytes

LED
light emitting diode

LFRC
low frequency R/C oscillator

LMA
load memory address

LSB
least significant bit

MPU
memory protection unit

MSP
main stack pointer

N.B.
note well — an abbreviation for the Latin term, *nota bene*

NMI
non-maskable interrupt

PC
program counter

POR
power on reset

PSP
process stack pointer

PWM
pulse width modulation

RAM
random access memory

RTC
real time clock

SDK
software development kit

SWO
software output — an output facility on Cortex-M devices

TCM
tightly coupled memory

UART
universal asynchronous receiver-transmitter

VLA
variable length array — a `C99` addition

VMA
virtual memory address

XT
crystal oscillator, typically of high accuracy

Appendix B

Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source code which is then given to the `micca` program. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the code in an order suitable for a language processor. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunk definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is in an acceptable order.

Appendix C

Target Platform

Background Information

- ARMv7e-M architecture
- “C” programming language

Microcontroller architecture

Microcontroller architecture

Apollo 3 Blue SOC

Apollo 3 Blue SOC

SparkFun Micromod Board

SparkFun Micromod Board

Tool chain

- Particulars of gcc and binutils

J-Link Debugging Probe

J-Link Debugging Probe

Other Segger Components

- RTT
- Debug monitor

Appendix D

Copyright Information

The following is copyright and licensing information. for the original material in this book

```
<<copyright info>>=
/*
 * This software is copyrighted 2021 - 2022 by G. Andrew Mangogna.
 * The following terms apply to all files associated with the software unless
 * explicitly disclaimed in individual files.
 *
 * The authors hereby grant permission to use, copy, modify, distribute,
 * and license this software and its documentation for any purpose, provided
 * that existing copyright notices are retained in all copies and that this
 * notice is included verbatim in any distributions. No written agreement,
 * license, or royalty fee is required for any of the authorized uses.
 * Modifications to this software may be copyrighted by their authors and
 * need not follow the licensing terms described here, provided that the
 * new terms are clearly indicated on the first page of each file where
 * they apply.
 *
 * IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
 * OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
 * THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
 * IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
 * NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
 * OR MODIFICATIONS.
 *
 * GOVERNMENT USE: If you are acquiring this software on behalf of the
 * U.S. government, the Government shall have only "Restricted Rights"
 * in the software and related documentation as defined in the Federal
 * Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
 * are acquiring the software on behalf of the Department of Defense,
 * the software shall be classified as "Commercial Computer Software"
 * and the Government shall have only "Restricted Rights" as defined in
 * Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
 * the authors grant the U.S. Government and others acting in its behalf
 * permission to use and distribute the software in accordance with the
 * terms specified in this license.
 */
```

The following is the copyright information for those parts of the code in this book which were directly or indirectly derived from the Ambiq Software Development Kit (SDK).

```
<<ambiq copyright info>>=
//*****************************************************************************
//
// Copyright (c) 2020, Ambiq Micro, Inc.
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions are met:
//
// 1. Redistributions of source code must retain the above copyright notice,
// this list of conditions and the following disclaimer.
//
// 2. Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
// documentation and/or other materials provided with the distribution.
//
// 3. Neither the name of the copyright holder nor the names of its
// contributors may be used to endorse or promote products derived from this
// software without specific prior written permission.
//
// Third party software included in this distribution is subject to the
// additional license terms as defined in the /docs/licenses directory.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.
//
//*****************************************************************************
```

Appendix E

Edit Warning

We want to make sure to warn readers that the source code is extracted and not manually written.

```
<<edit warning>>=
/*
 * DO NOT EDIT THIS FILE!
 * THIS FILE IS AUTOMATICALLY EXTRACTED FROM A LITERATE PROGRAM SOURCE FILE.
 */
```

Index

S

W